Grado en Ingeniería Informática

2016/2017

Trabajo Fin de Grado

Comparison of image classification with Shallow vs. Deep Neural Networks

Axel Blanco Cerro

Tutor

Juan Manuel Alonso Weber

2 de Octubre

uc3m | Universidad Carlos III de Madrid

Índice

Índice de figuras

Summary

Introduction

In the recent years there has been a boom in the ICT field of technologies related to Machine Learning and Data Science, in general. Gartner already included them as Smart Machines at the Top 10 Strategic Technology Trends For 2014 list [1].

That year several movements of big companies like Google or Facebook happened in those directions: at 2014's Google I/O conferences as Nvidia Toolkits, processing of data in the cloud or prediction technologies took place. Facebook also spoke about automatic tagging of photos or people recognition in their F8 conference.

Gartner –Figure 1.1–, in their 2013's analytics of emerging technologies, placed some disciplines related to Machine Learning, such as Speech Recognition, Predictive Analytics or Virtual Assistants. Moreover, they were placed beyond the peak of inflation expectations. It was estimated that it would take less than five years for their stabilization and market inclusion.

The artificial intelligence field –in general– and Machine Learning –for particular– are extremely broad fields and have a potential growth yet to be discovered. As I mentioned above, there are lots leading technological companies focused on developing technologies and services on top of these disciplines. Not only Google, Uber, Facebook or Netflix; even digital products companies still almost considered as startups –or maybe just fancy companies–, and also more traditional companies like IBM or Microsoft are betting very strongly on these new technologies.

Particularly, there has been a great amount of movements focused on the field of Deep Learning. In 2014 Google bought DeepMind, a startup specialized in it . Also in 2011 they published proprietary framework 'DistBelief' from Google Brain –the research division in Deep Learning– . It was later released in 2015 as free software, under the name of TensorFlow.

Uber, for example, has also extensively researched Deep Learning for their autonomous car project. Also Facebook: in the recent years has studied in depth image recognition technologies or labeling techniques. Facebook actually has presented an interior location technology based on image recognition with a mobile camera at F8 conference this year.

Academically speaking, the growth of interest in Neural Networks and Deep Learning

has increased exponentially in the last few years. This is partly motivated by improvements in infrastructure –better machines or processing with graphic cards– but also by the growth of frameworks and libraries to work with Neural Networks, as they really ease experimentation. It should also be noticed that great progress has been made due to the fact that large companies –such as Google or Facebook– have created whole departments dedicated exclusively to research and innovation in the Artificial Intelligence field.

But let's focus in the interesting stuff. This work intends to make a comparison between Shallow Neural Networks and Deep Neural Networks. As I mentioned before, in recent years the increase in interest in the so-called Deep Learning has been exponential. What is it? The answer to that question is not straightforward, and may require to clarify what Shallow Neural Networks are.

Shallow Neural Networks is a term that corresponds to a type of traditional Neural Networks. It is a term that includes networks with multilayer neurons, which usually have few layers –only one or two hidden layers–. These are networks with an indeterminate number of neurons in each layer, and usually use sigmoidal or tangential activation functions. This concept is an antithesis of Deep Neural Networks, which refer to networks with multiple layers that form part of Deep Learning techniques.

Along this project, we will discuss a comparison between Shallow Neural Networks and Deep Neural Networks, in order to understand and define the Deep Learning frontier. We have attended to a really rise of Deep Learning and Deep Neural Networks this last year: lots of big companies, as Google or Facebook, have opened their own research departments specialised in Deep Learning –sometimes it has been included in a Machine Learning or AI department–.

There are some interesting questions about Deep Learning and Shallow Neural Networks that call my attention. What type of problems must be resolved by a Deep Neural Network? What is the importance of GPUs in networks learning? What is the impact of networks architecture? Some of this questions are already solved, but others have no answer by now.

With these ideas on mind, the study will be oriented to study networks parameters and to find the differences between problems that make relevant the use of Shallow or Deep Neural Networks. It is also a good opportunity to make a small preliminary study about GPUs and their relevance in Deep Learning. This project allows us to use GPUs during all the experimentation and to be able to learn and share as they are configured and used in

this context.

The domain chosen for the tests has been image recognition. It is a very studied field from the Neural Networks prism –and many others, but we won't cover them in this project– and a large variety of datasets. Almost all datasets that have been analysed for experimentation are very studied datasets with much literature on the subject. This makes them very interesting for this work, since it allows focusing efforts on studying the behavior of networks rather than obtaining better or worse results. In fact, the MNIST, one of the sets that are analysed, is considered the Deep Learning '`hello world`'.

Bringing forward what can be studied in later sections, MNIST and CIFAR have finally been chosen as datasets on which to carry out the study. This domains are a good choice, as they are canonical sets that represent the difference between 'easy problem' and 'difficult problem', facilitating the comparison.

The MNIST dataset has, in its training set, 60000 images of 28 by 28 pixels size, reaching the 10000 images in the test set. The image size is 20 by 20 pixels centered, to which have been added a padding –up to 28 pixels– in order to facilitate their processing. In both datasets the number of classes is 10, since they represent the numbers 0 through 9. It is based on the much larger NIST dataset. The images included in MNIST have already been slightly preprocessed - centered and standardized - so that they have a common and standard format, without preprocessing needs by our side –if we do not want specifically–. In addition, the classes are balanced, which is interesting when studying the results, since we do not have deviations or strange conclusions derived from the imbalance.

The CIFAR dataset, on the other hand, is formed by small images –32 by 32 pixels– in color, labeled in ten classes according to the main element of the image –airplane, car, bird, dog, etc.–. This dataset has 50000 instances in the training set and 10000 in the test set. It is based on a much larger dataset called 'the 80 million tiny images', which is a set of small images representing almost 54,000 different concepts in English. From this dataset, a subset has been generated, with only some of the categories –ten, in particular– that have formed CIFAR10. Classes have been chosen so that they are totally mutually exclusive, that is, they can not have common or very similar elements. For example, this dataset cannot have a hawk class and an eagle one, just a bird class. This dataset is also balanced, which is especially useful and important for its use in more generalist experiments since it allows us to 'forget' the problem of unbalanced classes –that the majority class is always selected–.

TensorFlow and Keras are both selected as main frameworks, although several more

have been studied. The discussion can be found throughout the text.

TensorFlow [2] is a Machine Learning framework developed by Google and published in November 2015. It is a framework that pretends to be for general purpose –obviously, in Machine Learning topic– almost designed more as mathematics library, rather than as an abstract framework. It was originally created with the purpose of optimizing all Google's Machine and Deep Learning processes, although it was published as free software for community use. It is an extremely flexible framework, since it allows –and forces somehow– to build an operations graph that are intended to run in a completely personalized way –user can defined all operations and flows–. Once this graph is built, it is 'compiled' and executed with the input data provided. The compilation of this graph allows a very large parallelization, one of the main strengths of TensorFlow along with its support GPUs.

Keras [3] is a library focused on Deep Learning techniques, built on top of Theano or TensorFlow. It is a very powerful framework that uses Theano or TensorFlow engine, providing a great layers of abstraction with a lots of functions oriented to build simply and intuitive Neural Networks, or even used in Deep Learning. By using Theano, or TensorFlow underneath, you can extract the maximum power from each of them –by choosing between them as occasion demands– but with the ease of developing with a high level API. In addition, it is quite flexible with the definition of the Neural Networks, allowing to define the architecture of the network and the composition and parameterization of each layer as user wants.

Objectives

Due to the reasons that I have exposed in the introduction, the main objectives can be summarized in three of them:

- To study and analyze the impact of network and architecture parameters on Multilayer Perceptrons and Convolutional Neural Networks.
- To compare the performance between Multilayer Perceptrons and Convolutional Neural Networks in multiple image recognition problems.
- To study and analyze differences between Multilayer Perceptrons and Convolutional Neural Networks in order to decide what is better for each problem.

In addition, one interesting task is to define a problem frontier among Shallow Neural Networks and Deep Neural Networks. It looks like Deep Learning is a magical solution for all our problems, but at some jobs a Multilayer Perceptron or another traditional algorithm it is probably better.

One secondary goal is to achieve an in-depth understanding about Deep Learning and Deep Neural Netwoks, as I find it a very interesting topic.Furthermore, Data Science related jobs –in general– and Deep Learning jobs –for particular– have shown a spectacular rise in the labour market.

Finally, another secondary goal is to research about the main state of Neural Networks. Along this project, I will be able to study about particularities of Deep Learning as well as most powerfull and used frameworks. For this target, I will have planned to explore Theano and TensorFlow frameworks, but I would try to study other languages and frameworks as MXNet or CNTK, for example.

Experiments and experimentation

Throughout the experiments, I have chosen a Multilayer Perceptron architecture for Shallow Neural Networks representation, a Convolutional Neural Networks for Deep Neural Networks representation. Experiments have been structured in three blocks: one for Multilayer Perceptron parameters study, another one for Convolutional Neural Networks parameters, and last one for a comparison between this.

For Multilayer Perceptron architecture I have defined a traditional structure, also known as Feedforward Neural Networks. It is composed by three layers. The first layer is for network inputs –from MNIST or CIFAR–, with a sigmoid activation. The second one is the hidden layer. Note that I have changed the number of neurons throughout the experiments. For this layer I have tested sigmoid and ReLU activation functions. Finally, the architecture have one last layer with a softmax function and ten neurons. This is the output layer, and has one neuron for each class in order to select the output class. The softmax function –which behaviour has been discussed in Section 5.1– is an activation function designed for last layer for output normalisation.

Furthermore, the Convolutional Neural Network architecture have been cannonical too, although it have been more mutable than Multilayer. This network has two Convolutional layer and two dense layer plus one output layer. The output layer is like the Multilayer

Perceptron output layer: it has ten neurons –one for each class– with softmax as activation function. Dense layers are a fully connected layers, which has a variable number of neurons –same for each layer in our experiments– all of them connect with all neurons in the next layer. This layers are responsible of classification process. Usually each layer has a ReLU activation, but all of them are available.

Finally, the first two layers are Convolutional layers. This layer is a pretty special because is responsible of feature extraction, the big advantage of Convolutional Neural Networks. Each layer is, in fact, composed by two layers: one of them –first one– is the real Convolutional layer. This layer is able to recognize features from image. The other one is a pooling layer, which reduces input dimensions. In our architecture this layer use a max function –the layer is called max pooling layer– and reduce de dimensions by 2 –for example, a 4x4 matrix is reduced to a 2x2 matrix–.

I have experimented with network parameters in order to clarify the real impact that the have in network learning. I have used this experiments together with a more deeper research stage on Multilayer Perceptron and Convolutional Neural Networks afterwards. This analysis allowed me to reach a better understanding of the result of each experiment.

I have specifically modified the following parameters: learning rate, neurons for each layer, activation function and number of convolutional layers in Convolutional Neural Networks. A summary of most interesting experiments can be found below.

Learning rate The learning rate is a parameter that affects the stochastic gradient decrease. This gradient decrease reminds us that it is used to calculate the network learning, so that the error is optimized to minimize it. The learning rate regulates the speed of descent gradient convergation, to put it another way, the dimension of the 'jump' given by the error. It is closely related to the network's ability to avoid locals minimums, as with a high learning rate the 'jump' is higher and it makes it difficult to stop at a local minimum. Generally, with a high learning rate, larger oscillations of the error function occur with respect to the minimum. On the other hand, taking lower learning ratios of convergence, learning is usually slower and it is easier for learning to ground in local minimums.

Neuron's number The number and distribution of neurons in Neural Networks architecture is probably one of the most studied parameters of the networks. And also one of the parameters that more depends on the specific features of problem that is being tackled. In the study of these parameters it is an absurd to make any type of comparison between CI-

FAR and MNIST, since the choice of the number of neurons is going to be given by MNIST and CIFAR problems independently.

Activation function The activation function study has been one of the most surprising and interesting of the modified ones. In the first instance, the activation function of the network –in MNIST– did not seem too important, beyond most of the literature used sigmoids or tangential –hyperbolic or arcotangent– functions. However, given the little information about the reason of this choice, I have made some experiments in this regard. The study in CIFAR, on the other hand, a priori seemed to shed little light but it has noted the problem of over-training. In both cases, experiments of ReLU functions –which have been used by default in the CNNs– have been performed against the sigmoid function in the two dense layers.

Convolutional layers' number This section of the project is essentially motivated by the many references found in the literature regarding that with more than three or four convolutional layers, Convolutional Neural Networks began to works badly in this datasets. This could sound strange at first sight. In order to explain it, finding a balance in layers number is needed, just as a traditional Neural Networks does not improve the result by adding more layers indefinitely. However, Deep Learning techniques invite to think that the architecture should have many layers –thinking two or three as a reasonably small number of layers–, which is not necessary that way.

The conclusions of this section should be taken carefully, as the experiments carried out here only to correspond to a CIFAR domain –MNIST experiments have not been particularly significant–. This means that conclusions drawn here, even if they have a rather general character, are limited to the domain of CIFAR. For example, if three convolutional layers –with the particular configuration that I used– are too many layers in this case, does not mean that it is applicable to all convolutional networks.

Multilayer Perceptron versus Convolutional Neural Network From the MNIST experiments, the best results –in training– have been obtained with Multilayer Perceptrons. It is interesting that Convolutional Neural Network did not get to have a better result in training –apart from the experiment of Figure 5.13–. However, as a general measure of accuracy, Convolutional Neural Network are much more useful to approach the problem than Multilayer Perceptrons. Multilayer Perceptrons test accuracy –dark blue and green–

are well below of Convolutional Neural Networks test precision. It was already expected to achieve better results with Convolutional Neural Network, although the fact that training in Multilayer Perceptrons training sets works better in general is notable.

In CIFAR, on the other hand, it is completely different. Convolutional Neural Networks have a much higher accuracy than the Multilayer Perceptrons with an abysmal difference. On average, Convolutional Neural Networks achieve an accuracy of 0,8 approximately –in test–, while Multilayer Perceptron does not reach the average of 0,4.

Determined to set a threshold to know which one to use, with Convolutional Neural Networks, it seems that it is precisely the complexity and behaviour of the problem that can give us a clue. Of course, to confirm or deny any hypothesis in this area, the corresponding dose of experimentation should be performed. However, since CIFAR is a set of color images and MNIST does not, it seems that is one of the main differences. CIFAR encodes these colors so that it has a three-dimensional matrix, with the first two dimensions to indicate the height and width of the image, and the third to indicate the channel color. This type of structures –not exactly like that, but of that type– fit very well with Convolutional Neural Networks, since they receive as input directly that type of matrix, from which they extract the characteristics. On the other hand, networks with Multilayer Perceptrons architecture receive a single sequential vector.

In this way, any transformation of three-dimensional matrix to a one-dimensional vector will cause us to lose information. Maybe we can have same displaced positions, or maybe we have –if it transforms– the three consecutive channels. However, with inputs like the MNIST, which have a single color channel –grayscale–, we can transform it immediately to one-dimensional input, since the three colors will not be taken.

Results and conclusions

The conclusions of this comparison have been in line with expectations, based on the existing literature. On the one hand, the Convolutional Neural Networks have better performance, in general, than Multilayer Perceptrons, although in the MNIST dataset performance has been very close. This makes a lot of sense given the previous preprocessing layer that Convolutional Neural Networks have, and Multilayer Perceptrons does not have. Even in CIFAR, the difference of accuracy between Convolutional Neural Networks and Multilayer Perceptrons has been brutal.

The learning rate has proved to be a parameter, unnoticed initially, with an incredible impact on networks. As shown in Figure 5.9, learning rate influences can make a network learn absolutely nothing, despite having very good configuration in the rest of parameters. In addition, this parameter is closely related to the optimization function chosen in learning –Gradient descense, RMSprop, etc–, as well as to the shape and quantity of neurons in the layers. This makes it a simple parameter to modify, but with a very important weight in networks configuration. Also be careful with it, because learning rate is so linked to this other parameters that can give false negatives, causing a bad behavior in the network by changing these parameters independently.

Activation functions study has been quite interesting too, although the conclusions have not been as amazing. It has been possible to see how important this parameter was, since it has a great impact on the learning of the network. Convergence times, training and test accuracies, as well as general network learning are closely related to activation functions that are chosen.

However, the conclusions have been somewhat lukewarm as it has not yet approached a general rule. The most that has been deduced is that it is a parameter closely linked to optimization and error functions, as well as to the type of problem. For example, It is very usual to apply a softmax activation function in the last layer –the output layer– in classification tasks. This greatly helps the network 'to understand' results and learning better. Likewise, I have also seen that with very wide networks –many layers– or with Convolutional Neural Networks it is usual –and much more practical– to use ReLU functions instead of sigmoidal. This is because ReLUs have much less error attenuation with the passage of the layers in the backpropagation.

Special mention requires the network architecture, because it can not be considered a parameter itself, although it is especially influential for the adequacy of the network to the problem –as it seems obvious, on the other hand–. The network architecture –not only understood by itself is a convolutional network, a recurrent network or a traditional ones, but by the network structure itself– is a very important parameter to keep in mind. For example, once you decide to use a Convolutional Neural Network, it does not have the same impact or behavior that network has one convolutional layer that it has three of them; or that the number of neurons in the dense layers are the same or not.

These parameters greatly affect networks learning and are terribly linked to the type of problem, leaving only the opportunity to improve this aspect through experimentation

with the dataset.

However, I have reached some interesting conclusions: after some experiments modifying the amount of convolutional layers, it is seen that more does not always mean better. Specifically, from three convolutional layers –in CIFAR–, obtained accuracies normally fall sharply. This follows from the difficulty of propagating error backwards in this type of network –even with ReLU functions and special optimizers, such as RMSprop–. Just as interesting is the fact that more than two fully-connected layers, or even a neurons amount that are not equals in both layers, begins to produce a reduction in general network performance quite remarkable.

Consider all of this, in problems like MNIST we have to evaluate two objectives: on the one hand the research objective, according to which the best precision is sought, in which case Convolutional Neural Networks are the best choice. On the other hand, it is possible to contemplate that if objectives were more productive –to have a system in a real environment– Multilayer Perceptrons probably were more interesting, because they take much less to execute and require less resource although they have something less of precision. Finally, another factor that comes into play, beyond which architecture works better or worse objectively, is a design question: for what purpose am I designing this model? Do I want it to be the best? Or do I pretend to be the most efficient model? And the measures of precision and efficiency are determined by the problem that arises.

On the other hand, at a more concrete level, a rule can be defined to use Convolutional Neural Networks or not: Convolutional Neural Networks are applied in datasets for which information can be represented as a two-dimensional or three-dimensional matrix, while the Multilayer Perceptron can only understand one-dimensional inputs. This is because Convolutional Neural Networks receive as input a matrix, usually bi or three-dimensional. From this matrix can apply the convolutional layers, but without having this data structure, its performance suffers.

However, Multilayer Perceptrons receives one-dimensional vector as input. This means that the data must be able to be transformed into that shape in a coherent way. A typical solution is simply to concatenate the data of the two-dimensional array. This process –done this way– usually causes to break the spatial relationships in problem variables, making the work to the neural network more difficult. The more complex the problem and the more spatial relationships you have, the more impact you will notice in network accuracy.

Future work

After everything exposed in this work, it's quite clear that there is still much scope for improvement in this area. The issues raised have been resolved to a great extent, though. Neural networks in general, and Deep Learning in particular, presents major challenges for the scientific and research community nowadays.

Just a few years ago, access was very restricted and experimentation was very complex. In the current state of the art, the amount of tools and resources available for free, make possibilities almost infinite. Given this, the simplest access to relatively standard equipment, makes the research much easier even though it does not belong to any large research center.

Focusing on the work done, there are still many questionsto be answered. For example, one of the future experiments on a pending state, could be redoing all this experimentation with other types of sets - a priori, of image like the SVHN, but studying other areas could even be considered. Adapting this experimentation in other discipline datasets such as natural language processing or even time series prediction - a task in which traditional Neural Networks are reasonably good– could be extremely enriching. It would also offer new perspectives and bring new results on the matter.

On the other hand, even with the data sets used in this project, it would also be very interesting to test the effectiveness of other techniques. As an example, pre-processed with autoencoders instead of convolutional layers could be considered. The challenges and possibilities are many and very exciting. Deep Learning is still taking his first steps, so we still have to see its great rise.

1. Introducción

Este capítulo recoge las secciones introductorias de este trabajo fin de grado. Concretamente hablaremos de una descripción general del análisis realizado, seguida de una motivación inicial de la elección del tema y los objetivos que se pretenden alcanzar. Para finalizar se puede encontrar una sección donde se analiza la estructura utilizada en el documento.

1.1. Descripción general del trabajo

Este trabajo pretende realizar una comparación entre *Shallow Neural Networks* y *Deep Neural Networks*. En los últimos años el aumento del interés en el llamado *Deep Learning* ha sido exponencial. ¿Qué es? La respuesta a esa pregunta no es sencilla, y probablemente requiera previamente de aclarar el otro término de la ecuación: las *Shallow Neural Networks*.

Shallow Neural Networks es un término que corresponde al tipo de redes neuronas tradicional. El nombre viene del inglés, donde *Shallow* significa superficial, que se contrapone al *Deep* –profundo– de las *Deep Neural Networks*. *Shallow Neural Networks* es un término que abarca las redes de neuronas multicapa tradicionales, que generalmente tienen pocas capas –normalmente solamente una o dos ocultas–. Son redes con una cantidad indeterminada de neuronas en esa capa, y normalmente funciones de activación sigmoidales o tangenciales. En la Sección 2 "*Estado del arte*" pueden conocerse con mayor detalle. Y este concepto es una antítesis de las *Deep Neural Networks*, que hacen referencia a las redes con múltiples capas que se engloban dentro de las técnicas de *Deep Learning*.

La línea de delimitación de estas últimas –las *Deep Neural Networks*– es algo más difusa, pues no hay acuerdo sobre si deberían hacer referencia exclusivamente a las redes de neuronas con muchas capas –profundas–, o a todas las redes de neuronas que se integran dentro del *Deep Learning*, que no siempre tienen por qué tener forma de Perceptrón Multicapa. A efectos de este trabajo consideraremos esta segunda opción –todas las redes de neuronas en *Deep Learning*–, por tener mucho más sentido ya que la mayor parte de técnicas utilizadas en *Deep Learning* corresponden a CNNs o demás redes, casi nunca a redes de neuronas tradicionales con muchas capas.

Al estudiar la literatura del tema, muchas veces las elecciones de arquitecturas de

redes para según qué problemas se determina en base a la experiencia previa –bien propia o de la existente en el estado del arte actual–. Esta situación lleva a que muchas veces no se entiendan bien algunas de las elecciones, o sea difícil establecer unas guías básicas para la elección de las mismas. Esta problemática es la que motiva, en gran medida, este trabajo. Se pretende estudiar las características de las redes implicadas en ambas disciplinas –*Shallow Neural Networks* y *Deep Neural Networks*, sus diferencias, y que las hace interesantes para qué tipos de problemas.

Así mismo, es una buena oportunidad para realizar un pequeño estudio previo sobre las GPUs y su importancia en el *Deep Learning*. Una de las mayores problemáticas asociadas a las *Deep Neural Networks* deriva, precisamente, del coste computacional que conlleva asociado. Este coste es muy alto debido a la gran cantidad de operaciones matriciales complejas que implican la mayoría de las técnicas de *Deep Learning*, bien por complejidad de cálculos –funciones de activación o de optimización más complejas–, o bien por cantidad de capas. Las GPUs están especializadas en este tipo de cómputos, por lo que parecen una buena aproximación para solucionar esta problemática. También para usarlas durante toda la experimentación y poder aprender y compartir como se configuran y utilizan en este contexto. Igualmente se han escogido como *frameworks* con los que trabajar TensorFlow y Keras, aunque se han estudiado varios más, algo que puede encontrarse también la Sección 2 "*Estado del arte* ".

Con este estudio se busca encontrar una de esas guías de las que hablábamos anteriormente para la elección a priori de una arquitectura en base al tipo de problema que tiene. Una de las incógnitas, estudiadas en menor detalle, es por qué para un tipo de problema como el MNIST los MLPs función muy bien, y sin embargo para problemas como el CIFAR –del mismo tipo, pero más complejos– funcionan francamente mal. Esa es otra de las cuestiones que pretende dar respuesta este trabajo.

El dominio elegido para las pruebas ha sido el reconocimiento de imágenes. Es un campo muy estudiado desde el prisma de las redes de neuronas –y desde otros muchos, pero que no nos interesan tanto–, y una gran variedad de conjuntos de datos. Casi todos los conjuntos de datos que se barajan para la experimentación son *datasets* muy estudiados y con mucha literatura al respecto. Eso los hace muy interesantes para este trabajo, pues permite centrar los esfuerzos en estudiar el comportamiento de las redes más que en obtener mejores o peores resultados. De hecho, el MNIST, uno de los conjuntos que se barajan, se considera el "*hello world*" del *Deep Learning*.

Adelantando a lo que se podrá estudiar en secciones posteriores, se ha escogido finalmente MNIST y CIFAR como conjuntos sobre los que realizar el estudio. Son una buena elección pues son conjuntos canónicos que representan la diferencia "problema fácil" contra "problema difícil", facilitando el estudio.

Se han elegido para el estudio, por tanto, dos arquitecturas básicas, una de cada ámbito a estudiar. Un Perceptrón Multicapa –MLP– como representante de las *Shallow Neural Networks*, y una Red de Neuronas Convolucional –CNN– para las *Deep Neural Networks*. Estas redes son suficientemente representativas de cada ámbito para poder identificarlas con facilidad, y a la vez son muy útiles en la tarea elegida –el reconocimiento de imágenes–.

1.2. Motivación

En los últimos años, se ha producido un boom en el ámbito TIC de las tecnologías relacionadas con el aprendizaje automático y la ciencia de datos en general. Ya lo avecinaba Gartner en 2014 [1], donde se incluía lo que llamaban *Smart Machines*. También, ese mismo año, había movimientos de grandes compañías como Google o Facebook en esa dirección: en el Google I/O de 2014 ya se podían ver conferencias de Nvidia Toolkits, procesamiento de datos en la nube o técnicas de predicción. Además Facebook hablaba ya de etiquetado automático de fotos o reconocimiento de personas.

El propio Gartner –Figura 1.1– en su analítica de tecnologías emergentes ya situaba algunas disciplinas relacionadas con el *Machine Learning*, como por ejemplo *Speech Recognition*, *Predictive Analytics* o *Virtual Assistants*, pasado el pico de inflación de expectativas y estimando menos de cinco años para su estabilización e inclusión en el mercado.

El campo de la inteligencia artificial y, particularmente, del *Machine Learning* son extremadamente amplios y tienen un potencial de crecimiento aún por descubrir. Como he mencionado anteriormente, hay una gran cantidad de empresas tecnológicas punteras volcadas en desarrollar tecnologías y servicios usando estas disciplinas. No solo Google, Uber, Facebook o Netflix, empresas de productos digitales que casi todavía son consideradas *startups*, si no también empresas más tradicionales como IBM o Microsoft están apostando muy fuerte por estas nuevas tecnologías.

También en los procesos de transformación digital a los que se someten las empresas cada vez se incorporan más procesos inteligentes o automáticos para apoyar o simplificar los procesos de las mismas. En muchos casos no es más que una evolución o un caso particular

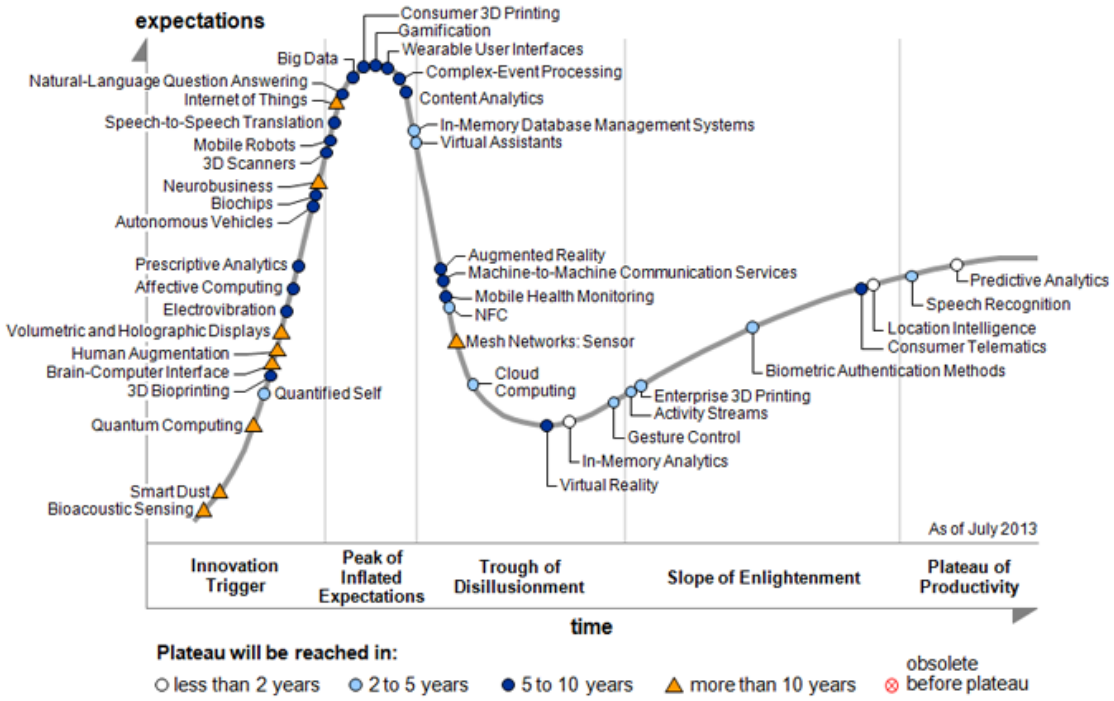


Figura 1.1: Hype Cycle for Emerging Technologies, 2014
Fuente: http://www.gartner.com/newsroom/id/2575515

en la aplicación de tecnologías de *Big Data* o *Bussines Inteligence*.

Particularmente, en el campo del *Deep Learning*, vemos como se ha producido una gran cantidad de movimientos dirigidos a la investigación en ese campo. En 2014 Google compró la compañía DeepMind, una startup especializada en deep learning. Así mismo, ya en 2011 sacaban desde Google Brain –la división de investigación en *Deep Learning*– el *framework* privativo DistBelief que se liberó posteriormente en 2015 bajo el nombre más conocido de TensorFlow. También Uber, por ejemplo, ha investigado ampliamente en *Deep Learning* para su proyecto de coches autónomos. O Facebook, que en los últimos años ha profundizado en las tecnologías de reconocimiento y etiquetado de imagen, llegando a sacar en su conferencia de 2017 una tecnología de localización en interiores basada en reconocimiento de imágenes con la cámara de un móvil.

Los campos del *Machine Learning* y *Deep Learning* son, por tanto, disciplinas que generan un gran interés a día de hoy en la industria. Tienen mucho aún que ofrecer y pueden implicar cambios significativos en la sociedad actual y en cómo nos comportamos o interactuamos con el mundo. Cómo se puede ver en la Figura 1.2, comparándola con la Figura 1.1,

Figura 1.2: Hype Cycle for Emerging Technologies, 2016
Fuente: http://www.gartner.com/newsroom/id/3412017

ya en 2013 Gartner daba mucha confianza a las tecnologías relacionadas con el *Machine Learning*. En 2016 podemos ver que esas disciplinas ya ni se consideran tecnologías emergentes y que el propio *Machine Learning* es una tecnología que se adoptará mayoritariamente en menos de cinco años, estando en la cresta de la ola.

Académicamente hablando el crecimiento del interés en redes de neuronas y *Deep Learning* en particular también ha aumentado exponencialmente en los últimos años. En parte motivado por las mejoras a nivel de infraestructura –mejores máquinas o procesamiento con tarjetas gráficas, también conocidas como GPUs–, pero también por el crecimiento de *frameworks* y librerías para trabajar con redes de neuronas y facilitar la experimentación con ellas. También cabe destacar que se han conseguido grandes avances gracias a que grandes empresas –como Google o Facebook– han creado departamentos enteros dedicados exclusivamente a la investigación e innovación en el campo de la inteligencia artificial.

Sin embargo, ya particularmente hablando del ámbito que nos ocupa –*Deep Neural Networks* y *Shallow Neural Networks*–, en muchas ocasiones es difícil encontrar explicaciones

al por qué de ciertas decisiones, ampliamente utilizadas en la bibliografía del tema, pero sin una motivación clara –más allá del mítico funciona–. En parte este vacío motiva este trabajo. Estudiar las diferencias entre *Deep Neural Networks* y *Shallow Neural Networks*, y tratar de poner en claro para qué son preferibles unas u otras. ¿Matar moscas a cañonazos o la única manera de obtener resultados?

1.3. Objetivos y alcance

Los objetivos de este trabajo se pueden resumir esencialmente en 3:

- Evaluar el impacto general de las diferentes características de una red de neuronas clásica con arquitectura *Multilayer Perceptron* –MLP en adelante– y *Convolutional Neural Network* –CNN en adelante–.
- Comparar el rendimiento –efectividad vs coste– de ambas arquitecturas en diferentes tipos de problemas de clasificación de imágenes.
- Estudiar las diferencias entre ambas arquitecturas que las hacen preferibles para uno u otro conjunto de datos.

Además de estos objetivos de tipo más general, se pretende encontrar o definir una frontera entre los distintos tipos de problema a partir de la cual es más conveniente utilizar técnicas de *Deep Learning* frente a técnicas más tradicionales con *Shallow Neural Networks*.

También se ha incluido, para dar rigor y consistencia al proyecto un presupuesto asociado así como una planificación de la distribución del trabajo. Al ser un proyecto de un carácter eminentemente práctico, no se ha considerado relevante especificar concretamente la metodología de desarrollo utilizada. Nótese que únicamente se ha desarrollado el código relativo a los experimentos –implementación de las diferentes arquitecturas– y los scripts necesarios para ejecutar los experimentos y visualizar los resultados. El código de las arquitecturas correspondientes puede encontrarse en los apéndices.

Avanzando ligeramente lo que se tratará en profundidad en la Subsección 3.1, solamente se abordarán dos conjuntos de datos, el MNIST [4], que está formado por imágenes de números manuscritos, y el CIFAR10 [5], formado por imagenes de un carácter más general. Con esto se pretende fijar algunas ideas y dar una visión global y general del tipo

de problema para el que puede resultar interesante el uso de *Deep Learning*, pero aún podría realizarse un estudio en mayor profundidad con conjuntos de datos más grandes o con diferentes características.

1.4. Estructura del documento

Para la consecución de los objetivos previamente marcados, el documento se distribuye de la siguiente forma:

En la sección actual, "*Introducción* ", se puede encontrar los objetivos generales del Trabajo Fin de Grado, así como su alcance y motivación. La intención principal de esta sección es que sirva de resumen y colección de ideas generales del camino que seguiremos a lo largo de todo el documento.

En la Sección 2 "*Estado del arte* " se incluye una reflexión e introducción más profunda a la temática que tratamos y al estado del arte en general. De esta forma, en esta sección el lector puede introducirse, de forma más general, al reconocimiento de imágenes o las redes de neuronas con las que trabajaremos. También se localiza en esta sección los marcos socio-económico y regulador que afectan al tema que se está tratando.

En la Sección 3 "*Diseño de la solución* " se han incluido de forma más específica los conjuntos de datos y *framework* escogidos, así como un análisis entre las alternativas existentes, justificando la elección realizada.

En la Sección 4 "*Instalación del entorno* " se puede encontrar una breve guía de instalación del software utilizado durante este trabajo. Concretamente la instalación de los drivers de NVIDIA, CUDA, cuDNN y los *frameworks* escogidos. La motivación de ello ha sido el coste que ha supuesto para mi la instalación del software, por lo que me ha parecido razonable agrupar todas las instrucciones en una pequña guía.

En la Sección 5 "*Experimentación y estudio* " se recoge el grueso del trabajo. Es donde se analizan y estudian todos los experimentos realizados, así como se plantean las arquitecturas finales utilizadas. Durante la descripción de la experimentación se abordan todas las cuestiones que me han parecido interesantes para abordar los objetivos planteados y construir las conclusiones del trabajo.

En la Sección 6 "*Conclusiones y trabajo futuro* " se pueden encontrar las conclusiones generales resultado del análisis de la experimentación, así como las lineas de trabajo futuro en las que se podría seguir investigando.

Por último, la Sección 7 "*Planificación y presupuesto*" está dedicada a la planificación del trabajo que se ha desarrollado, así como al presupuesto de los recursos empleados –tanto personales como de infraestructura–.

Adicionalmente se encuentra toda la bibliografía utilizada en el transcurso del documento. También se puede encontrar en los apéndices el código base empleado para la experimentación.

2. Estado del arte

2.1. Aprendizaje automático

El aprendizaje automático es una disciplina dentro de las ciencias de la computación que pertenece de la rama de la inteligencia artificial. Basa su trabajo en el desarrollo de técnicas que permitan a algoritmos autónomos aprender. Para ello se han desarrollado técnicas muy variadas –Redes de Neuronas, árboles de decisión, algoritmos Genéticos, maquinas de vectores de soporte... Las aplicaciones de este campo son muchas y muy variadas, yendo desde la detección de *spam* a la detección de tumores, pasando por la agrupación de sectores de la población, o el apoyo a la toma de decisiones.

Esta disciplina entiende que hay dos formas de abordar un problema en función del tipo de datos de los que se disponga:

Aprendizaje supervisado Es aquel aprendizaje que se realiza a partir de datos etiquetados. De esta forma, los algoritmos van afinando sus métodos de clasificación o regresión –las tareas más comunes– para responder a esas etiquetas. Una vez entrenado el sistema, se entiende que esta listo para recibir datos nuevos no etiquetados y etiquetarlos. Por ejemplo, se podría utilizar para abordar tareas como la clasificación de imágenes.

Aprendizaje no supervisado Es aquel aprendizaje que se realiza sobre datos no etiquetados. Generalmente aborda tareas de agrupación o de extracción de características comunes a esos datos, pues no puede –a priori– entender que etiquetas aplicar. Por ejemplo, es una forma de abordar problemas de agrupamiento de individuos.

Existe una tercera vía, el **aprendizaje semi-supervisado**, que está en desarrollo. Basa su funcionamiento en trabajar con un pequeño conjunto de datos etiquetas junto a una cantidad más grande de datos sin etiquetar, lo que parece que mejora su exactitud. Naturalmente, tiene sus propias técnicas que no necesariamente tiene que ser las mismas que al usar aprendizaje supervisado o no supervisado.

En el presente trabajo se va a realizar un estudio centrado en las redes de neuronas con una tarea de aprendizaje supervisado para usarlo como elemento conductor en el análisis que se pretende realizar. Por eso no se invertirá más tiempo en desglosar toda la información

de la que se dispone sobre aprendizaje automático, aunque sin duda es un campo apasionante en todos sus aspectos.

2.2. Redes de neuronas

Una red de neuronas es un modelo computacional englobado dentro de las técnicas de aprendizaje automático. La primera referencia que tenemos a una red de neuronas propiamente dicha –aunque sin una implementación, pues es una aproximación matemática– se remonta a 1943, cuando el neuropsicólogo Warren McCulloch y el matemático Walter Pitts proponen el primer modelo matemático que aproxima una red de neuronas. Años más tarde, en 1958, Frank Rosenblatt plantea el primer modelo concreto basado en una neurona de McCulloch-Pitts.

Llegó a implementarlo en una máquina –antes de que existieran siquiera lenguajes de programación– que llamó Mark I Perceptron. Formó parte de un proyecto del ejercito estadounidense con el laboratorio de aeronautica Cornell [6], pensado para extraer características de una imagen de 400 píxeles –20x20–. Este proyecto salió adelante y fue un éxito, al menos dentro de lo que las tareas que se querían realizar.

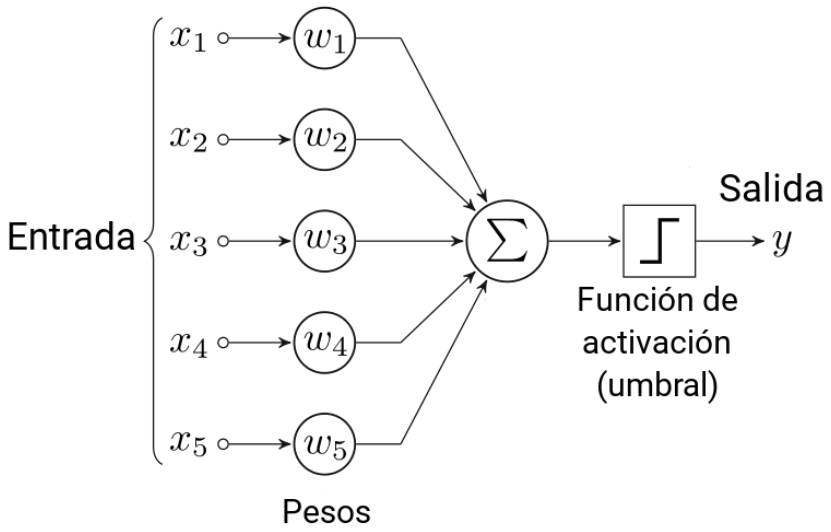


Figura 2.1: Perceptrón simple basado en la neurona de McCulloch-Pitts
Fuente (modificada): https://commons.wikimedia.org/wiki/File:Perceptr%C3%B3n_5_unidades.svg

Lo llama Perceptrón, y es un sistema integrado por una única neurona que estima una salida binaria en base a una entrada y un umbral. Como podemos ver en la Figura 2.1, la idea es muy sencilla. Se realiza el sumatorio de todas las entradas por unos pesos y se utiliza el umbral para devolver la salida binaria. La ventaja que presenta este sistema –el Perceptrón– es que asocia un método de aprendizaje. Por medio de la presentación de patrones y resultados previamente conocidos pueden modificarse esos pesos de forma que el sistema aproxime una función lineal –puesto que, en el fondo, no resulta más que una combinación lineal de entradas y pesos– que divide el espacio del problema en dos "clases", que corresponden a la salida binaria del Perceptrón. La función de modificación de los pesos depende del error de clasificación, de forma que se va ajustando en base al error de clasificación de la neurona.

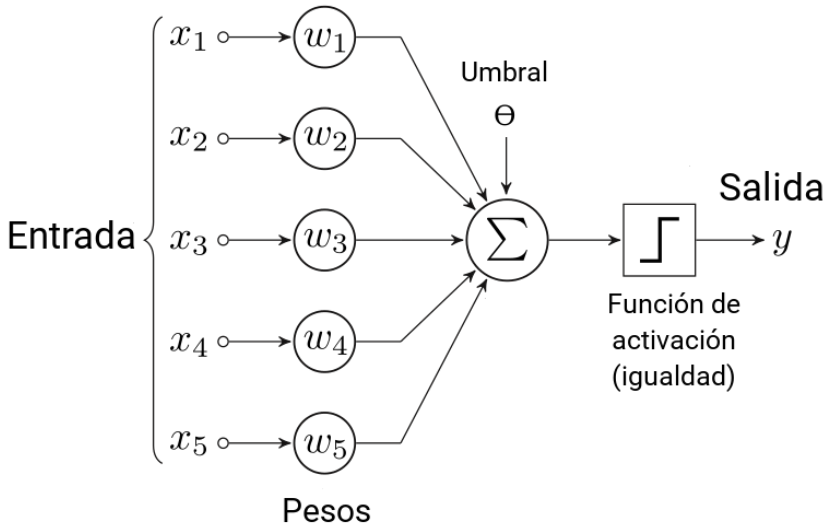


Figura 2.2: Adaline basado en la neurona de McCulloch-Pitts
Fuente (modificada): https://commons.wikimedia.org/wiki/File:Perceptr%C3%B3n_5_unidades.svg

Algo más tarde, en 1960, Bernard Widrow y Ted Hoff desarrollaron otro sistema basado en la neurona de McCulloch-Pitts, que llamaron Adaline (ADAptative LINear Element). Este sistema tiene una forma similar al Perceptrón, que puede verse en la Figura 2.2. La diferencia esencial reside en la función de activación, aunque también se añade una componente más al sumatorio en forma de umbral, que representa típicamente el bias estadístico. La función de activación en este sistema pasa a ser la identidad, por lo que la salida del mismo

es el sumatorio de la propia entrada por los pesos más el *bias*.

Este modelo tiene un funcionamiento muy similar al Perceptrón, salvo porque realiza aproximaciones lineales directas, es decir, estima funciones lineales como si de una regresión lineal se tratara. No es un modelo demasiado usado, aunque puede ser útil, puesto que existen problemas de regresión o de predicción que modelan funciones lineales o casi lineales, por lo que el Adaline –que a todos los efectos genera una regresión lineal– es muy apropiado.

2.2.1. Arquitectura básica

Tanto el Perceptrón simple como el Adalaine son perfectamente válidos como primera aproximación, y tienen aplicación en una amplia variedad de problemas. Sin embargo, para aproximaciones más complejas, por ejemplo, para aproximar la función XOR, no son válidos. Al ser su salida lineal, aproximan razonablemente bien muchos problemas –lineales o muy cercanos a la linealidad–, pero fracasan estrepitosamente con problemas no lineales complejos. Al ser este tipo de problemas precisamente un conjunto muy amplio de los que se pretenden resolver en el mundo real, hubo que buscar alternativas a estos modelos.

Esto es demostrado en 1969, por Minsky y Papert, quienes además argumentan que estos dos sistemas no son generalizables en redes compuestas –Minsky hablaba particularmente del Perceptrón, pero es aplicable al Adaline también–. Esto se debe a que si se compone una red con varios Perceptrones, por la forma que se tiene de calcular el error –que es unitario por cada neurona–, no se podría calcular de forma sencilla cómo afecta el error de cada neurona en el resto de capas, al menos en el momento actual.

Para ilustrar un poco las líneas anteriores, y también para poner en claro como se plantea un Perceptrón Multicapa –MLP, MultiLayer Perceptron–, podemos atender a la Figura 2.3. Como se puede ver, la idea principal detrás de una arquitectura MLP, de las más extendidas dentro de las redes de neuronas, es apilar Perceptrones en capas sucesivas. La primera capa, la capa de entrada, típicamente hace referencia a los propios valores de entrada. La capa oculta –o las capas ocultas– contiene los Perceptrones, operando la salida de la capa anterior por los pesos. Y la capa de salida contiene las salidas de la red, que normalmente representan las clases en las que se pretende clasificar o el resultado de la regresión –depende mucho de cada problema–. El cálculo de la salida de una neurona sería entonces:

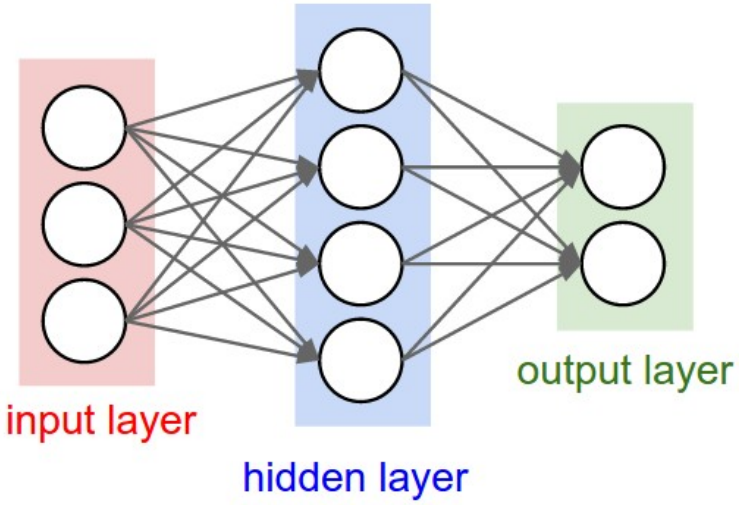


Figura 2.3: Perceptrón multicapa
Fuente: http://cs231n.github.io/neural-networks-1/

$$a_i^c = f(\sum_{j=1}^{n^{c-1}} w_{ij}^{c-1} a_j^{c-1} + u_i^c) \; para \; i = 1, 2, ..., n_c \; y \; c = 2, 3, ..., c-1$$

Donde n_c corresponde a la cantidad de neuronas en esa capa –de forma que i las recorre todas–, $c-1$ hace referencia a la capa anterior –de forma que w_{ij}^{c-1} y a_j^{c-1} corresponde a los pesos y la activación en la capa anterior– y u_i^c al bias de la neurona actual.

Es necesario hacer notar que al construir un MLP los Perceptrones sufren una ligera transformación conforme a los Perceptrones simples que vimos anteriormente. Esta es que la función de activación, en lugar de ser la identidad o una función de umbral, puede tomar diferentes valores de funciones no lineales. Típicamente las utilizadas son la función sigmoidal logarítmica, la tangente hiperbólica o la arcotangente, aunque también se utilizan otras funciones de activación como la ReLU –Rectified linear unit– y sus derivadas, sinusoidales o gaussianas; incluso funciones de activación pensadas exclusivamente para la capa de salida, como la función softmax.

Esto solucionaba la problemática de la aproximación de las redes de neuronas a funciones no lineales, pero sigue dejando pendiente solucionar el proceso de aprendizaje, que sigue sin funcionar en este tipo de redes. Si se calcula el error en las neuronas de la capa de salida, podemos actualizar los pesos de la última capa hacia la capa de salida, pero en esos momentos no sabían como actualizar los errores del resto de neuronas.

2.2.2. Regla Delta Generalizada

Fue precisamente esta situación la que hizo perder la fe a parte de la comunidad científica en este tipo de técnicas, relegándolas a investigaciones residuales durante los años setenta. A pesar de todo, se siguió investigando en el tema, y en 1974 Paul Werbos propone la regla de retropropagación –*backpropagation*–, que se basa recorrer la red de neuronas de forma normal, hacia delante –*feedforward*–, comparar el resultado esperado con el obtenido, y propagar ese error hacia atrás, actualizando cada peso en base a su capa anterior y el error que hasta ella se ha propagado. Esta teoría, sin embargo, pasó desapercibida para la comunidad hasta que David Rumelhart, Geoffrey Hinton y Ronald Williams publicaron en 1986 un articulo en Nature llamado "Learning representations by back-propagating errors" [7]. En este articulo postulaban un *framework* construido sobre la teoría de Werbos donde explicaban cómo realizar exactamente esa propagación hacia atrás.

Así surgió la Regla Delta Generalizada. La idea es realizar un descenso de gradiente sobre la función de error, propagando el gradiente a las neuronas de capas anteriores de forma que se pretende minimizar la función de error. Con ello y con la regla de la cadena se generan las ecuaciones para actualizar los pesos. Con esta regla ya se puede entrenar de forma efectiva una red de neuronas de varias capas, y se abrió un nuevo camino para el estudio de las redes de neuronas.

La solución al problema del entrenamiento de las redes viene acompañada de la publicación de "Multilayer feedforward networks are universal approximators" [8]. En este artículo, publicado en 1989, demuestra matemáticamente que las redes de neuronas multicapa –concretamente, los MLPs– pueden aproximar teóricamente cualquier función. Esta demostración es reveladora, pues confirma que las redes de neuronas pueden, virtualmente, resolver cualquier problema. Naturalmente, la efectividad de una red concreta para aproximar una función concreta no necesariamente es del 100 %.

2.3. Deep learning

Las primeras referencias al término *Deep Learning* se dan a finales de los años ochenta, cuando Rina Dechter por una parte presentó el término ante la comunidad, y LeCun por otra parte empezó a construir redes de neuronas MLP con muchas capas. Como disciplina dentro del aprendizaje automático, el *Deep Learning* abarca todas aquellas técnicas que agrupan varias capas con funciones de activación no lineales y que incluyen células de

extracción de características.

En general, al trabajar con técnicas de *Deep Learning* se asume que cada capa añade una capa más de "abstracción" sobre el problema, siendo técnicas muy interesantes para tratar problemas complejos o de muy alto nivel. Por ejemplo, reconocimiento de imágenes, reconocimiento de voz o procesado del lenguaje natural. Además, tiene otra ventaja muy interesante en algunos ámbitos; al incluir en muchas ocasiones capas de "extracción de características", como por ejemplo las capas convolucionales, se puede obviar el trabajo previo de ingeniería de características.

A pesar de que se lleva hablando de *Deep Learning* desde los años 80, la verdadera explosión de estas técnicas se ha producido en los últimos años. En parte se debe a la cantidad de datos disponibles actualmente, nada comparables con los que había hace años. Para que estas técnicas exploten su máximo potencial, deben alimentar con una cantidad ingente de datos de los que no se disponía hasta estos últimos años. Además, está la cuestión del procesamiento, pues muchas técnicas de *Deep Learning* –sobre todo arquitecturas grandes o complejas– tienen un coste computacional elevado. Las maquinas de que se disponía no eran suficientemente potentes, y no ha sido hasta estos últimos años con la explosión de potencia en las GPUs que se ha podido avanzar en esa linea.

2.3.1. Redes de Neuronas Convolucionales

Las Redes de Neuronas Convolucionales se diferencian de las redes de neuronas tradicionales en una primera capa de extracción de características que incluyen al inicio de la red. Como se puede ver en la Figura 2.4, la red cuenta con una o más capas convolucionales –en este caso dos– antes de tener una o más capas totalmente conectadas al estilo tradicional.

Esta composición que puede parecer algo ajena inicialmente, tiene mucho sentido cuando se estudia en detalle. En primera instancia, nos ahorramos un proceso previo de extracción de características –*feature extraction*–, que es muchas veces tedioso y propenso a errores. Las capas convolucionales realmente constan de dos capas cada una, una capa donde se realiza la propia extracción de características y una segunda capa donde se realiza un muestreo, reduciendo la cantidad de variables a la mitad, manteniendo las componentes más importantes. La primera capa de extracción de características realmente lo que consigue es dividir la imagen en trozos más pequeños de forma que cada neurona de la capa convolucional puede centrarse en extraer un tipo de característica concreta. El doble objetivo final de todo este proceso es poder reducir los tamaños de las matrices de pesos –que en problemas

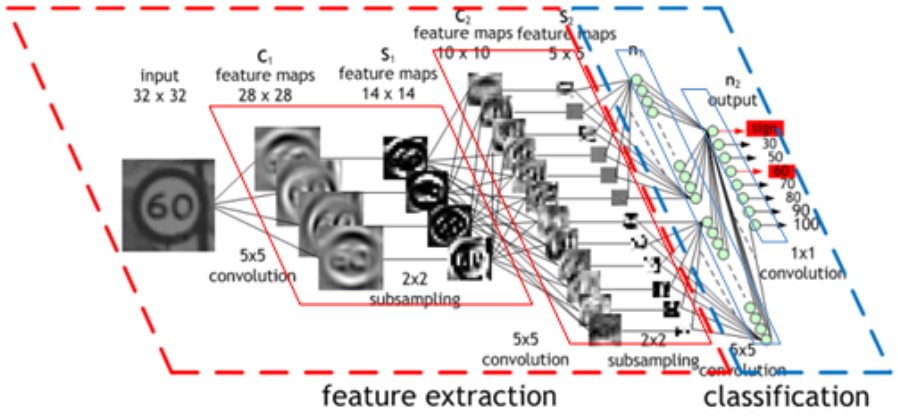


Figura 2.4: Red de Neuronas Convolucional
Fuente: http://www.andreykurenkov.com/writing/a-brief-history-of-neural-nets-and-deep-learning-part-2/

complejos pueden ser cuantitativamente grandes– a la vez que se mejora la funcionalidad de la red pues también es capaz de aprender qué características escoger.

Las últimas capas, las que son totalmente conectadas, normalmente son capas tradicionales. En ocasiones se varia la función de activación para adecuarla al tipo de problema o arquitectura de la red. Por ejemplo, en la literatura se encuentran referencias a que en CNNs la función de activación ReLU funciona mejor que la sigmoide debido al desvanecimiento del descenso de gradiente que sufre esta última. La salida de la red es igual que en cualquier otra red de neuronas, suele representar las clases a clasificar o el resultado de la regresión.

Resulta bastante interesante contrastar que el uso de las redes convolucionales está bastante extendido al usar *Deep Learning* en general, no solo para reconocimiento de imágenes. Las primeras capas de extracción de características resultan funcionar muy bien también con otro tipo de tareas, como el reconocimiento de voz.

2.3.2. Otras arquitecturas usadas en Deep Learning

Como las técnicas de *Deep Leaning* sea usan en muchas más aplicaciones a parte del reconocimiento de imágenes, hay otras arquitecturas que se usan bastante para otro tipo de tareas. A nivel informativo –aunque no se han utilizado en este trabajo– mencionaré dos de las más conocidas.

2.3.3. Redes de Neuronas Recurrentes

Las Redes de Neuronas Recurrentes –RNN por sus siglas en inglés– son un tipo de red de neurona que basa su funcionamiento en que la salida de la red alimenta como entrada en la siguiente iteración de la misma. Su uso y estudio se popularizó en torno a los años 80 ya que las redes de neuronas "hacia delante" no podían solucionar cierto tipo de problemas. Concretamente, aquellos cuyo patrón en cada estado dependía de los estados anteriores. Por ejemplo, este es un problema reiterado en la disciplina de NLP –*Natural Language Processing*, Procesado del Lenguaje Natural– puesto que para entender frases humanas no basta con analizar cada palabra, si no que es importante tener en cuenta el resto de palabras de la frase.

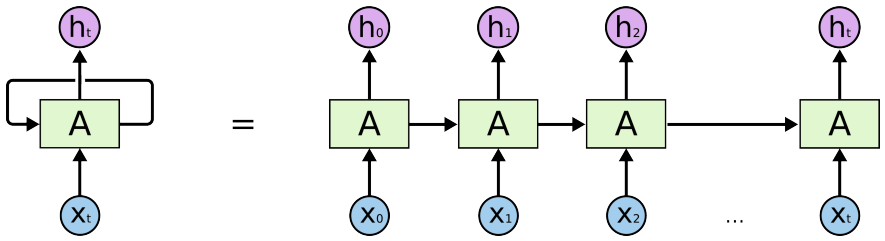


Figura 2.5: Red de Neuronas Recurrente
Fuente: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

En la Figura 2.5 podemos ver la arquitectura de una Red de Neuronas Recurrente tipo. Aparece primero en su forma normal, y después "desenrollada" para poder ilustrar la idea detrás de estas reglas. Como decíamos anteriormente, la salida de la primera iteración h_0 sirve de entrada a la activación de la neurona recurrente en la iteración siguiente, cuando la entrada sea x_1.

Con el paso del tiempo se han diseñado diferentes arquitecturas de redes de neuronas recurrentes, en las que se varía su activación, la forma en que se comunica la recurrencia, etc. Cada una de ellas se ha ido especializando en un tipo de problemas concreto. Por ejemplo, encontramos redes LSTM –de las que hablaremos a continuación con algo más de detalle–, recursivas, jerárquicas, redes de Hopfield, etc.

2.3.4. Redes de Neuronas Recurrentes Temporales

El problema que presentan las Redes de Neuronas Recurrentes es que de haber dependencia entre la salida y la entrada con varios niveles de recurrencia intermedios empieza

a funcionar mal. Por ejemplo, si la salida $h2$ que vemos en la Figura 2.5 depende de la entrada $x0$, nuestra red recurrente probablemente nunca llegue a detectar esa relación. Para paliar ese efecto surgieron las *Long Short Term Memory* –LSTM–, un tipo de red recurrente cuya activación es más compleja que la activación de las redes recurrentes normales.

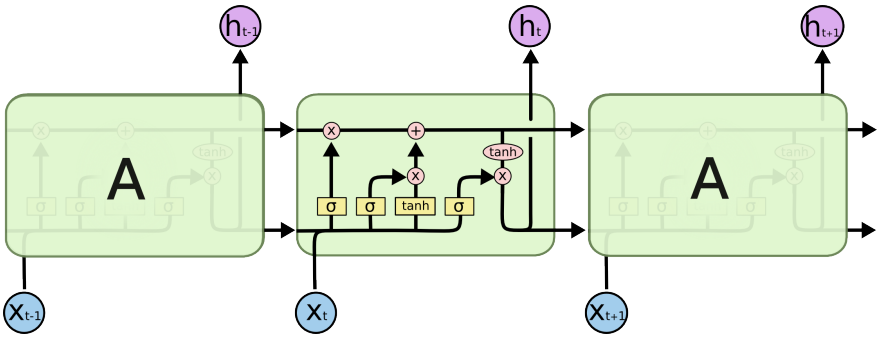


Figura 2.6: Long Short Term Memory
Fuente: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Esta activación, como se puede ver en la Figura 2.6, consta de varias operaciones, no solo de la aplicación de una función sobre el sumatorio de la entrada, y ofrece dos salidas en lugar de una, que alimentan la recurrencia. Con estas dos salidas se puede generar cierta "memoria" para que la red pueda detectar estas relaciones entre términos que se encuentren a larga distancia, temporalmente hablando.

Las aplicaciones de las LSTMs son amplias, y casi todas tienen que ver con aquellos campos o problemas en los que tener una memoria sea importante. Por ejemplo, las LSTMs se ha demostrado que funcionan muy bien en problemas de reconocimiento de voz, de generación de señales –por ejemplo, musicales– o en cierto tipo de problemas de predicción de series temporales. Son redes que para funcionar bien necesitan muchos datos –en general, con *datasets* pequeños no son demasiado eficientes– y cuyo coste computacional es alto.

Para conocer más sobre el funcionamiento concreto de estas redes el enlace de fuente de la Figura 2.6 es especialmente interesante, pues ofrece una introducción bastante acertada.

2.4. GPUs y su impacto en el Deep Learning

Históricamente uno de los factores limitantes del Aprendizaje Automático en general, y del *Deep Learning* en particular ha sido el tiempo de computo. Generalmente los tiempos de entrenamiento de modelos de Aprendizaje Automático para problemas reales suelen ser

bastante altos, sobre todo si se cuentan con muchos datos. Hasta hace poco, los algoritmos de *Deep Learning*, si bien era muy potentes e impresionantes, no resultaban competitivos pues los tiempos de entrenamiento –y muchas veces de ejecución– eran excesivamente altos. Naturalmente, esto teniendo en cuenta que eran tareas realizadas por CPUs cuya potencia no era demasiado elevada. Ni siquiera los científicos con acceso a sistemas HPC –*High Performance Computing*– obtenían tiempos mucho mejores. Debido a esto la investigación en este ámbito estaba bastante limitada.

El problema proviene, principalmente, del tipo de tareas en las que funciona muy bien una CPU. Incluso con CPUs potentes como tenemos ahora –con muchos *cores*, y mucha velocidad–, la principal cuestión radica en que las CPUs se pensaron –y se usan– como máquinas con pocos *cores* capaces de llevar a cabo acciones complejas. Son maquinas de propósito general, diseñadas para realizar operaciones muy complejas, pero no especialmente cálculos muy complejos.

Por otra parte, las GPUs se diseñaron específicamente para realizar operaciones matemáticas matriciales muy complejas por medio del uso de muchos núcleos que realizasen operaciones muy simples. No en vano fueron pensadas originalmente para el procesado de gráficos, que implica una cantidad de operaciones algebraicas sobre matrices bastante alta. De esta forma, el tiempo que tarda una GPU en ejecutar el entrenamiento de un algoritmo –que es, esencialmente, operaciones con matrices y cálculos avanzados– se reduce drásticamente. Cabe destacar que las GPUs han evolucionado mucho, e incluso los propios fabricantes diferencian ya entre GPUs pensadas para trabajar con gráficos y GPUs pensadas para trabajar con algoritmos de *Machine Learning* o computación más intensiva.

El uso de GPUs combinadas con CPUs no se empezó a dar en la comunidad científica hasta 2001. Por ejemplo, NVIDIA no es hasta 2006 que publica CUDA, un lenguaje y unas herramientas pensado para desarrollar soluciones explotando al máximo la GPU. Esto es lo que se viene usando de un tiempo a esta parte: el uso de CPUs para los procesados más generalistas –carga de datos, preprocesado básico, escritura en disco– y combinarlo con la utilización de GPUs para las tareas más intensivas matemáticamente hablando.

2.4.1. CUDA

Como ya se mencionaba anteriormente, CUDA fue publicado por NVIDIA en 2006 como una suite de herramientas para los desarrolladores que usaban NVIDIA –que incluía un lenguaje propio basado en C con su compilador correspondiente–. La intención es poder

programar desde C o C++ software paralelizado para ser ejecutado en la GPU con una interfaz directa a la misma.

Desde su publicación, los campos de aplicación de las GPUs han crecido enormemente. Al poder ser usado por el público en general, gran cantidad de investigadores o de científicos han podido implementar sus experimentos o soluciones en estas tarjetas gráficas –GPUs–. Esto ha contribuido enormemente a que el mundo de *Deep Learning* tenga el impacto que tiene hoy dia.

cuDNN Sobre CUDA NVIDIA ha construido cuDNN, que es una librería que incluye las directivas más usadas en *Deep Learning*. En cuDNN se implementan esas directivas sobre CUDA directamente, optimizando drásticamente las ejecuciones. Estas directivas pueden ser invocadas desde todos los *frameworks* que utilizan CUDA, ya que la mayoría lo usan como conector.

La estructura básica con la que se suele trabajar cuando se plantea el uso de GPUs se puede ver en la Figura 3.4. Generalmente se tiene la capa de driver –típicamente NVIDIA, pero puede ser otro–, sobre esta se construyen las librerías especializadas de ese driver. Como se puede ver en la Figura 3.4 –mencionada anteriormente, capas dos y tres de nuestro diagrama– en este caso se trata de CUDA y sobre ella cuDNN. Finalmente ya se situa el *framework* a utilizar en la última capa, que conectará con todo lo anterior.

Todo esto es una muestra más del avance que está sufriendo el mundo de las GPUs y como el *Deep Learning* se ha convertido en un nicho de mercado muy importante para todas las empresas que trabajan en este ámbito. Toda esa inversión e innovación repercute en hardware cada vez más potente y una cantidad de librerías software cada vez más grande, con muchas soluciones particulares, por ejemplo, en el mundo de las redes de neuronas.

2.5. Marco regulador

Para la redacción de este TFG se ha tenido en cuenta el marco legal que afecta a cada una de sus partes. Se detalla a continuación el impacto de la legislación vigente tanto en el desarrollo del propio trabajo como en las posibles aplicaciones del mismo –que se estudiarán con mayor detalle en la sección siguiente.

Por un lado hay un impacto inicial de las licencias del software utilizado –tanto *frameworks* y lenguajes como bases de datos–. En este sentido la elección de los mismos

también venía influida por este motivo. El lenguaje y *frameworks* escogidos –y, en general, casi todos los estudiados– tienen una licencia de software libre de algún tipo.

En concreto, la licencia de Python –Python Software Foundation License (PSFL) [9]– es de tipo software libre permisiva, aunque de elaboración propia –por la Python Software Foundation–. Es compatible con GPL, y cumple con las cuatro libertades, aunque permite la licenciación de las modificaciones tanto bajo licencias libre como bajo licencias privativas. Keras está licenciado bajo MIT License [10], una de las licencias de software libre menos restrictivas existentes. TensorFlow también tiene una licencia de software libre, en concreto la Apache License 2.0 [11], que es algo más restrictiva, en concreto es limitante en cuanto a la transmisión de la marca –obras derivadas no podrán usar la marca TensorFlow– y a la garantía y responsabilidades –TensorFlow no se hace responsable de ninguna forma de obras derivadas–.

Respecto a los conjuntos de datos, MNIST, que es propiedad de Yann LeCun (Courant Institute, NYU) y Corinna Cortes (Google Labs, New York), permite la descarga y uso libremente citando la fuente [4]. La misma situación se da con CIFAR, que pese a no tener licencia –aplica el copyright usual–, Alex Krizhevsky, Vinod Nair y Geoffrey Hinton, sus creadores, indican al distribuirla que se puede usar libremente siempre y cuando se cite el informe técnico en el que lo publicaron inicialmente [5].

Por otro lado, veo necesario examinar el marco legal de las posibles aplicaciones del presente trabajo. En este caso, como veremos más adelante, no hay demasiadas aplicaciones directas, al ser un trabajo de estudio de tecnologías existentes. Sin embargo, en cuanto se sale de los conjuntos de datos académicos y se pretende utilizar las tecnologías aquí estudiadas en proyectos con impacto real, hay dos cuestiones legales bastante importante que considero inevitable estudiar.

Dado que el estudio realizado en este TFG se centra en el reconocimiento de imágenes, es relevante recordar la posible violación de privacidad derivada de las imágenes que se traten. Por ejemplo, en un sistema de reconocimiento de rostros o de patrones de comportamiento en base a vídeos o imágenes, es muy importante que la obtención de esos recursos gráficos esté dentro de la legalidad y no se haya realizado sin el consentimiento o conocimiento de los filmados. No solo en la obtención de conjuntos de entrenamiento y test, si no en el uso productivo del modelo generado. Conviene recordar la LOPD –Ley Orgánica de Protección de Datos–, ya que muchas de las violaciones de privacidad vienen derivadas del desconocimiento e incumplimiento –con conocimiento de causa o sin él– de esta norma.

Así mismo, tenemos también una responsabilidad ética sobre los modelos que se generen de nuestro trabajo, así como la aplicación final que tenga. En este sentido, debemos ser muy cautos, pues estas tecnologías –la inteligencia artificial en general y las redes de neuronas en particular– pueden rápidamente utilizarse para acciones innobles, como el control de la población, la vigilancia de los ciudadanos o la predicción de comportamientos. Está en nuestras manos asegurarnos de que sea bueno o malo para la sociedad.

Naturalmente, también es importante tener en mente las implicaciones legales de la Inteligencia Artificial en su conjunto, así como lo joven que es la legislación respecto a las nuevas tecnologías en general, y a los algoritmos inteligentes en particular. Por ello, cabe reflexionar sobre las posibles ilegalidades o alegalidades que se puedan derivar de ello, aunque es un debate que posiblemente escape mucho del alcance de este proyecto.

2.6. Entorno socio-económico

La naturaleza de este trabajo –experimental, pero teórico– hace que sea algo complejo establecer un impacto directo del mismo en la sociedad. Por ello, hablaremos de los usos que pueden tener las tecnologías con las que se trata en el proyecto, más que el impacto del trabajo propiamente dicho.

En sí mismo el trabajo pretende poner en valor las características principales que diferencian –y hacen mejores para según que problemas– las técnicas utilizadas. Esto puede ser interesante para el conjunto de la comunidad en tanto en cuanto que puede dar pistas a la hora de aproximarse a un problema sobre la técnica que probablemente funcione mejor –cabe recordar que en esta disciplina la técnica que mejor funcione depende en gran medida del problema concreto–.

Más allá de eso, los campos de aplicación de las técnicas de *Deep Learning* son muy variados. En el caso particular que se ha utilizado para este estudio, la aplicación más directa es el reconocimiento facial o reconocimiento de imágenes, pero hay muchos otros ámbitos en los que puede resultar particularmente interesante: el etiquetado de vídeos, descripción automática de imágenes, reconocimiento de patrones imágenes –por ejemplo, de melanomas–.

Y, por supuesto, las técnicas basadas en redes de neuronas tienen una gran variedad de aplicaciones, desde predicción de comportamientos –más o menos complejos–, conocimiento de patrones de voz, comprensión del lenguaje natural. También permite mejorar los sistemas de apoyo a la toma de decisiones, lo cual puede optimizar los procesos que se siguen

actualmente en una gran cantidad de campos, como la medicina. Así mismo, es muy útil en técnicas de predicción o en sistemas financieros –aunque los resultados no son extraordinarios, debido a la gran cantidad de variables existentes.

Todas estas aplicaciones permitirían –y permiten, las que ya hay implementadas– ganar mucha eficiencia en procesos repetitivos o que requieren de cierto grado de inteligencia, permitiendo al usuario disponer de ese tiempo para dedicarlo a otras labores. Por utilizar un ejemplo clarificador, una de las aplicaciones más extendidas para la inteligencia artificial son los filtros de *spam* en los *email*. Un filtro de *spam* que use técnicas inteligentes tiene una efectividad muy alta, y reduce los tiempos que dedica el usuario a revisar correo basura en gran medida.

3. Diseño de la solución

En esta sección se analizarán las diferentes posibilidades de conjuntos de datos, lenguajes de programación y librerías que se podrían utilizar. Durante el análisis se irán perfilando las decisiones que se tomen para definir las herramientas con las que se trabajará. No se ha incluido ningún análisis sobre las diferentes arquitecturas a tratar. Esto se debe a que el objetivo del trabajo es analizar el comportamiento de las arquitecturas MLP y CNN, por lo que cualquier otra alternativa queda fuera del alcance del proyecto. De todas formas, en las secciones correspondientes a la arquitectura MLP y a la CNN de la experimentación se analiza la arquitectura concreta escogida.

3.1. Conjuntos de datos

Los conjuntos de datos que se han estudiado para realizar la selección han sido conjuntos de datos de imágenes. Dado que el dominio del problema es de reconocimiento de imágenes se han considerado exclusivamente conjuntos de datos de este ámbito, intentando también que fueran de acceso libre y contasen con una comunidad amplia

3.1.1. MNIST

El conjunto de datos MNIST [4] –*Modified National Institute of Standards and Technology database*– es una base de datos de dígitos manuscritos con una gran cantidad de datos, muy utilizada en tareas de procesado de imagen en general. En el ámbito del reconocimiento de imágenes casi se considera el *Hello world*.

Este conjunto de datos tiene, en su conjunto de entrenamiento, 60000 imágenes de 28 por 28 píxeles de tamaño, alcanzando las 10000 imágenes en el conjunto de test. Realmente son imágenes de 20 por 20 píxeles centradas a las que se les ha añadido un *padding* –hasta los 28 píxeles– para facilitar su procesado. En ambos conjuntos el número de clases es de 10, ya que representan los números del 0 al 9.

Está basado en el conjunto de datos NIST, mucho más grande. Las imágenes incluidas en MNIST ya han ligeramente preprocesadas –centradas y normalizadas–, de forma que tienen un formato común y estándar, sin ser necesario un preprocesado. Además, las clases están balanceadas, lo que es interesante a la hora de estudiar los resultados, ya que no con-

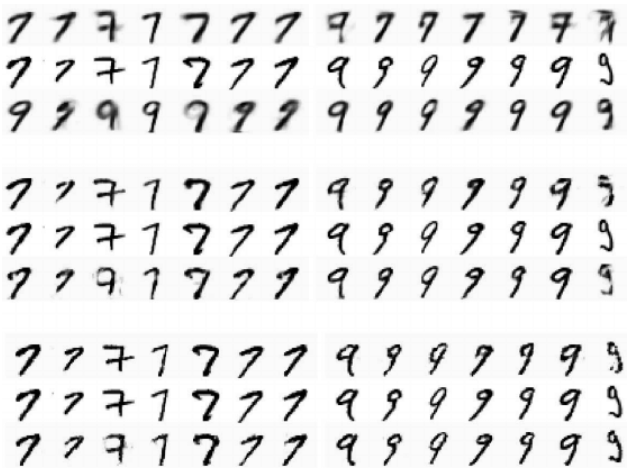


Figura 3.1: Ejemplo de 7s y 9s en el MNIST
Fuente: https://www.researchgate.net/profile/Guy_Mayraz/publication/2540340/figure/fig4/AS:279612832665609@1443676302993/Fig-5-Cross-reconstructions-of-7s-and-9s-with-networks-of-25-100-and-500-units-png

tamos con desviaciones o conclusiones extrañas derivadas del desbalanceo. Concretamente cuenta con –en su conjunto de entrenamiento– 30.000 datos del conjunto SD3 –*Special Database 3*– de NIST, y otras 30.000 imágenes del conjunto SD1 –*Special Database 1*–, así como 5.000 y 5.000 de cada conjunto en su conjunto de test. En total calculan que hay dígitos de unos 250 autores distintos, no solapándose los autores del conjunto de test con el conjunto de entrenamiento.

Esto es especialmente interesante ya que en la base de datos original –NIST– se concibió el conjunto SD3 como conjunto de entrenamiento y el conjunto SD1 como test. Esta decisión vino motivada por que el conjunto SD3 proviene de número manuscritos por empleados de la oficina del censo estadounidense, mientras que el conjunto SD1 proviene de números escritos por estudiantes de instituto. Naturalmente, la legibilidad del primer conjunto –SD3– lo hace más deseable para un conjunto de entrenamiento debido a su claridad, legibilidad y consistencia –entre cada una de las imágenes recogidas–. Sin embargo, al estar mezclados en el MNIST, hace que el conjunto de entrenamiento sea algo más complejo, permitiendo que los sistemas inteligentes –en nuestro caso, las redes de neuronas– sean capaces de generar modelos más generalizables.

Es el conjunto de datos más sencillo de los que se plantea utilizar, está ampliamente estudiado y actualmente se tiene tan precisión sobre él que, en el conjunto de test, hay siste-

mas que solo fallan 170 imágenes –98.3 % de acierto–. Es una buena opción para valorar las redes de neuronas tradicionales, ya que en el resto de conjuntos los resultados que obtienen son poco alentadores. A la vez, se puede comparar su resultado con las arquitecturas convolucionales, ya que el margen de mejora en este conjunto es relativamente pequeño, y valorar la eficiencia de ambas. También es interesante el hecho de que las clases estén balanceadas y ya se incluya un pequeño preprocesado inicial, que facilita nuestro trabajo sobre este *dataset*.

3.1.2. CIFAR10

El conjunto de datos CIFAR10 [5] es un subconjunto de otro *dataset* mucho mayor que consta de imágenes de pequeño tamaño –32 por 32 píxeles– en color, etiquetadas en diez clases según el elemento principal de la misma –avión, automóvil, pájaro, perro, etc–. Este conjunto de datos cuenta con 50000 instancias en el conjunto de entrenamiento y 10000 en el conjunto de test.

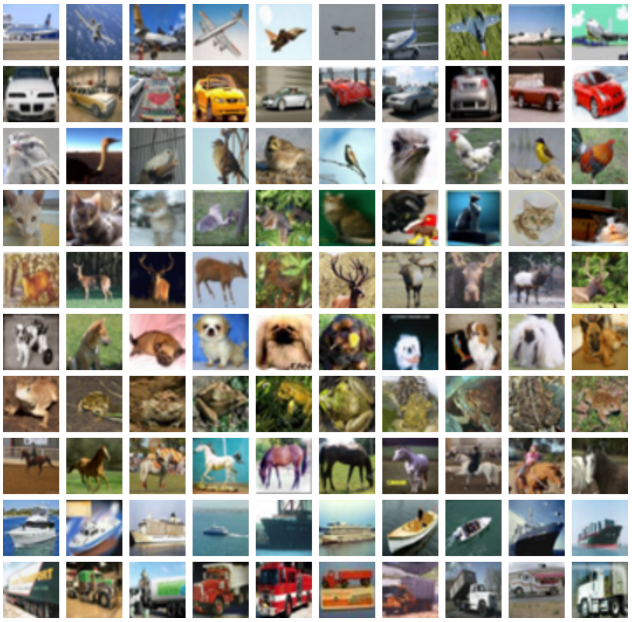


Figura 3.2: Ejemplo de imágenes de CIFAR10
Fuente: https://www.cs.toronto.edu/~kriz/cifar.html

Está basado en otro conjunto de datos mucho mayor, el *80 million tiny images*, que es un conjunto de imágenes de pequeño tamaño que representa casi 54.000 conceptos diferentes en inglés. De este conjunto se ha generado un subconjunto, con solo algunas de las categorías –diez, concretamente–, que ha formado el CIFAR10. Las clases han sido escogidas

de forma que sean totalmente excluyentes entre sí, es decir, que no puedan tener elementos comunes o muy parecidos y están balanceadas. Esto último es especialmente útil e importante de cara a su uso en experimentos más generalistas ya que nos permite "olvidar" el problema de clases desbalancedas –que se prime siempre la clase mayoritaria–.

Es un conjunto de datos mucho más complejo, no solo en tamaño, si no en estructura. Tiene, por ejemplo, tres canales de color –es RGB-, o elementos más complejos que números manuscritos, presentes en el mundo. Es, aún así, un conjunto de datos muy utilizado, ya que permite probar la potencia de las redes convolucionales en poco tiempo, comparado con conjuntos de imágenes más grandes –en tamaño de la imagen–.

También existe el conjunto de datos CIFAR100, con la misma estructura, pero con 100 clases en lugar de 10. Este conjunto, sin embargo, es más complejo de tratar y se aleja un poco del alcance del problema –es más costoso de trabajar–. Al ser ambos conjuntos iguales variando únicamente la cantidad de clases, me he decidido por priorizar el CIFAR10, que puede ser mucho más útil para el proyecto que pretendo llevar a cabo. También cabe resaltar que las 100 clases del CIFAR100 están agrupadas en superclases y no están tan claramente diferenciadas, lo que dificulta el problema.

3.1.3. SVHN

El conjunto de datos SVHN [12] que contiene imágenes recortadas de Street View donde se ven únicamente los números de los portales o edificios. Cuenta con unas 600000 imágenes, de las cuales unas 70000 forman el conjunto de entrenamiento y unas 20000 el conjunto de test, mientras que el resto se ofrecen para que el investigador las distribuya como prefiera.

Es un conjunto de datos muy generalizado también, que ofrece la posibilidad de realizar una tarea de reconocimiento de dígitos (como el MNIST), pero mucho más compleja, pues son imágenes de 32 por 32 pixels donde los dígitos no están aislados, si no en un entorno real. Además las imágenes son en color. Tiene dos representaciones diferentes, de las cuales la segunda sería interesante para nosotros. La primera muestra la imagen del número de portal completo, permitiendo reconocerlos en su contexto, mientras que el segundo formato aplica un preprocesado similar al MNIST, normalizando los números y aislándolos. Esto facilitaría enormemente nuestra tarea.

Está a medio camino entre la complejidad de CIFAR10 y MNIST, ofreciendo un mayor margen de mejora del que ofrece MNIST, pero manteniendo el reconocimiento de

Figura 3.3: Ejemplo de imágenes de SVHN
Fuente: http://ufldl.stanford.edu/housenumbers/examples_new.png

dígitos, que es un problema más sencillo que el reconocimiento de carácter más general.

3.1.4. ImageNet

El conjunto de datos ImageNet [13] es un conjunto de datos que aglutina imágenes a tamaño real, organizado siguiendo la definición de WordNet. Esto quiere decir que el etiquetado de las imágenes se hace en base a la jerarquía definida por WordNet, lo que es especialmente útil para descripción automática de imágenes o si se pretende extraer información semántica de la misma.

La cantidad de imágenes por clase varía en función de la misma, desde las cientos de imágenes en algunos nodos semánticos –etiquetas– hasta las miles de imágenes en algunos otros. Así mismo, el formato no es consistente ya que las imágenes son de tamaños y formas muy variadas. También es importante tener en cuenta que las imágenes que contiene ImageNet no son de su propiedad, es decir, esta base de datos solo almacena la referencia a su localización y, en ocasiones, una miniatura de la misma, pero los derechos de copyright y de uso los tiene quien publicara la imagen. Esto implica también que los formatos de las imágenes (png, jpg...) pueden ser muy diferentes entre cada una de ellas.

Además, este conjunto de datos cuenta con algo más de 14 millones de imágenes de diferentes tamaños y formas, lo que le hace especialmente útil o interesante en caso de pretender entrenar un sistema productivo que se utilice en entornos reales.

Sin embargo, tras analizar en detalle el conjunto, escapa un poco de los límites del presente proyecto. No solo en cuanto a forma y tamaño del conjunto –que es demasiado

grande y quiere demasiado procesado previo para ser interesante–, si no en cuanto a enfoque y finalidad, pues este conjunto esta pensado para tareas mucho más de descripción automática o reconocimiento de imágenes complejas, ámbitos que no abordaremos.

3.1.5. Elección final

Finalmente utilizar los conjuntos MNIST y CIFAR10 es la opción más razonable. Como ya he mencionado anteriormente, son conjuntos de datos "canónicos" dentro de los problemas de reconocimiento de imágenes, están ampliamente estudiados y casi todos los *frameworks* o librerías tienen interfaces o documentación suficiente sobre cómo procesarlos y trabajar con ellos.

Son además dos conjuntos suficientemente representativos de dos dominios de problemas diferentes, de una parte el reconocimiento de dígitos o caracteres manuscritos, y de otra el reconocimiento y etiquetado de imágenes de carácter más general. Este hecho es muy beneficioso para poder estudiar más a fondo los problemas que presentan según que tipos de arquitecturas al procesar cada uno de estos dos "tipos de problemas".

En caso de utilizar tres conjuntos (en lugar de dos como he definido para no abarcar demasiado) el conjunto SVHN también sería una opción muy asequible e interesante, que puede ofrecer conclusiones bastantes útiles para el problema que aquí se trata. Si finalmente no lo he escogido ha sido exclusivamente por la restricción autoimpuesta de utilizar solamente dos conjuntos de datos y el hecho de que CIFAR10 ofrece, en mi opinión, más posibilidades de explorar un problema complejo de forma sencilla.

3.2. Lenguajes de programación

Para la elección de lenguajes y *frameworks* a utilizar se ha realizado un pequeño estudio sobre las posibilidades tanto de lenguajes como de librerías o *frameworks* asociados a esos lenguajes. Es relevante recordar que el propio desarrollo de los experimentos no es en si mismo un fin del presente trabajo –si un efecto secundario deseado, un aprendizaje interesante–. Por este motivo, la elección del lenguaje y *framework*, tras el estudio de las posibilidades, ha venido determinado más por intereses personales y conocimientos previos que por una necesidad real del proyecto.

Como ya se ha manifestado previamente, las GPUs tienen una importancia vital dentro de las técnicas que utilizan *Deep Learning*. Principalmente debido a la potencia que

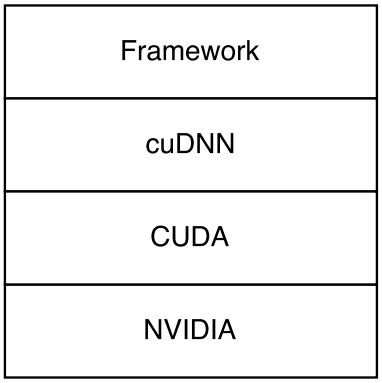


Figura 3.4: Pila de librerías y drivers utilizados en el desarrollo de los experimentos

ofrecen, que se traduce en unas reducciones de tiempos de cómputo muy interesantes. Por ello, se ha planteado para el desarrollo de los experimentos la utilización de una pila de librerías y drivers concreta, que puede encontrarse en la Figura 3.4. Aunque ya se ha descrito previamente, concretado a nuestra situación la pila sería de la siguiente forma: Como driver básico contamos con NVIDIA sobre una de sus tarjetas gráficas –la GTX 1050 4G–. Por encima tenemos una capa de CUDA, las herramientas de desarrollo de NVIDIA para sus gráficas. En el siguiente estrato –el tercero por abajo– tendremos cuDNN, el conjunto de directivas de *Deep Learning* de NVIDIA para CUDA. Esta librería será usada por la última capa, donde tendremos el *framework* escogido.

Cabe destacar, en cualquier caso, que aunque hay algunas características particulares que deben ser valoradas de cara a la elección de *framework*, realmente no han tenido una gran significancia –muchos *frameworks* de los estudiados cumplían ampliamente estas condiciones–. Repasemos dichas condiciones para facilitar el estudio de los siguientes puntos de la sección –siempre con el diagrama de la Figura 3.4 en mente:

El *framework* debe soporta CUDA y cuDNN Este es el punto más "crítico" de todos. El soporte de CUDA es más una condición deseable que un imperativo, pero facilitaba mucho el desarrollo del proyecto. Los tiempos de ejecución de los experimentos frente a las ejecuciones con CPU se reducen notablemente, lo que permite realizar más experimentos y ajustar más los tiempos. En muchas ocasiones estamos hablando de mejoras de más de 10 veces –como se puede ver en el Cuadro 3.1, extraída de la bibliografía [14], tabla 7. Además, por supuesto, ofrece conocimiento sobre el uso de una tecnología muy importante en grandes procesos de *Deep Learning*, como es la computación con GPUs.

Cuadro 3.1: Comparativa entre CPU y GPU (tiempo por minibach en segundos)

	Server CPU (Threads used)				Single GPU	
	1	2	8	32	G980	G1080
FCN-S	9.571	6.569	1.710	0.630	0.060	0.048
AlexNet-S	5.741	4.216	1.160	0.962	0.059	0.042
FCN-R	3.410	2.541	0.661	0.325	0.033	0.020
AlexNet-R	6.124	4.229	1.396	0.971	0.227	0.317

Por tanto, durante la selección buscaba un *framework* que fuese compatible con el procesamiento en CUDA y con la librería de NVIDIA cuDNN. Esta última es una librería que proporciona NVIDIA para proveer de primitivas muy usadas en las *Deep Neural Networks* sobre la capa de CUDA. Hay varios *frameworks* que soportan la integración con esta librería, lo que permite explotar al máximo las características de las tarjetas gráficas NVIDIA para este tipo de tareas.

Estudié también la posibilidad de utilizar OpenCL, aunque no está tan optimizado para este tipo de tareas como CUDA, a pesar de ser una alternativa libre. Esto se refleja en que la mayoría de los *frameworks* no implementan OpenCL o están empezando a desarrollar la implementación actualmente.

El *framework* debe poder ejecutarse desde una consola o en un script Esta condición –que puede parecer una perogrullada– es bastante relevante a la hora de automatizar la experimentación y reducir los espacios improductivos. Algunas de las herramientas inicialmente exploradas permitían realizar redes de neuronas mediante interfaces o en plataformas en la nube–como es el caso de Weka, o productos comerciales como "IBM SPSS Neural Networks"–, permitiendo una abstracción de gran parte de los problemas más programáticos, pero limitando en gran medida las posibilidades de experimentación.

Mediante un *framework* construido sobre un lenguaje de programación de uso general, que pueda ser ejecutado en cualquier máquina –aunque principalmente en Linux–, tenemos mucha más libertad y flexibilidad. Permite parametrizar el *script* para automatizar las ejecuciones en *batches* de experimentos, mejorando la eficiencia de los mismos. También nos permite realizar algunas abstracciones, pudiendo generalizar operaciones de las redes de neuronas y evitar duplicar esfuerzo y trabajo en la generación de redes parecidas.

El *framework* debe aportar flexibilidad y abstracción En linea con lo mencionado anteriormente, es deseable que el *framework* aporte algunos grados de abstracción a la hora de generar las redes. La suficiente abstracción como para no invertir tiempo en el desarrollo de funciones ya implementadas por la comunidad, pero en grado máximo que permita la modificación de los parámetros que deseamos modificar y estudiar a lo largo de este trabajo. Es decir, de nada sirve un *framework* que simplifica infinitamente el desarrollo de las redes, si a cambio perdemos la posibilidad de cambiar –y, por tanto, poder estudiar– alguno de los parámetros que nos resultan interesantes.

Este asunto es algo delicado ya que, a priori, el conjunto de los parámetros a estudiar puede estar definido de cierta forma, pero ir variando con la observación de los mismos, surgiendo nuevas lineas de estudio.

3.2.1. Lenguajes de programación

En general, el lenguaje de programación escogido para la realización del TFG es algo que ha venido supeditado a condiciones secundarias, ya que no tiene una importancia primordial para la realización del trabajo.

El análisis lo he realizado sobre tres lenguajes de programación diferentes: `C++`, `Python` y `R`. Inicialmente también contemplé `Java` y `Go` como unas opciones a considerar, aunque quedaron descartados en el estudio inicial. Esencialmente los motivos por los que descarté estos lenguajes son dos: por una parte, la cantidad de *frameworks* enfocados *Deep Learning* no son demasiados, y los existentes son poco flexibles; por otra parte, ambos lenguajes son algo más complejos de utilizar para la realización de *scripts*, ya que están más enfocados al desarrollo de software completo.

`C++` [15] Fue uno de los primeros lenguajes a analizar, a pesar de mi poco conocimiento sobre él. Es un lenguaje muy rápido en ejecución, para el que encontramos varios *frameworks* –por ejemplo, TensorFlow tiene API en `C++`, también encontramos OpenNN–. Ofrece como ventaja, esencialmente, que junto con la integración con CUDA permitía una ejecución muy rápida, reducir los tiempos de experimentación y toda la potencia de un lenguaje de propósito general.

Sin embargo, presentaba algunas dificultades. El primer punto en contra es que para explotar todo eso se debía tener un conocimiento razonablemente amplio del lenguaje y de cómo funciona, algo que no se cumple en mi caso. Además, hay que añadir las dificultades

derivadas de un lenguaje algo desconocido para mi con unas librerías sin un uso masivo –gran parte de la comunidad usa `Python` o `R` para la realización de experimentos–. Cabe destacar que, si bien la mayoría de las librerías más usadas están escritas o tienen interfaz para `C++` –Caffe, TensorFlow, CNTK...–, una gran parte de la comunidad usa las interfaces en otros lenguajes, principalmente `Python`, lo que dificulta en ocasiones la resolución de dudas o el mantenimiento.

Es un lenguaje altamente recomendable para el despliegue de modelos productivos, pues su eficiencia y velocidad supera con creces la de otros lenguajes de *scripting* –como `Python` [16]–, pero no resulta demasiado interesante como lenguaje para realizar experimentos. Es más lento de desarrollar que `Python` o `R`, y la cantidad de librerías y desarrollo previos es mínima en comparación con estos. Esto lo hace un lenguaje poco interesante para el tipo de tarea que se lleva a cabo en este trabajo.

`Python` [17] es un lenguaje de propósito general muy utilizado en *scripting*, pero que en los últimos tiempos tiene cada vez más uso para aplicaciones productivas. Es un lenguaje interpretado, que puede ser imperativo u orientado a objetos, con un tipado dinámico. Esta características –sobre todo el hecho de que sea interpretado– hacen que sea muy lento en comparación con otros lenguajes compilados –como veíamos anteriormente al compararlo con `C++`–.

Sin embargo, es un lenguaje que ha sido adoptado por gran parte de la comunidad, como un lenguaje menos académico –y más potente– que `Matlab` o `R`. Al ser un lenguaje de propósito general permite combinar los modelos o prototipos realizados con software más genérico, o incluso utilizar las librerías de las que dispone `Python` –que tiene una amplia variedad–. Por ejemplo, con `Python` interactuar con una base de datos es relativamente sencillo, y cuenta con librerías y conectores para la mayoría de las bases de datos más usadas. Esto ayuda a almacenar gran cantidad de datos en bases de datos en lugar de en ficheros, y poder recuperarla con más facilidad que otros lenguajes de *scripting*, cuya conexión con bases de datos es más difícil

Cuenta con bastantes librerías científicas que permiten trabajar con estructuras de datos más propias de `Matlab` o de `R` que de `Python`, e integrar todo eso con la gestión de un lenguaje bastante ágil y sencillo de desarrollar, sin demasiadas complicaciones –gracias al tipado dinámico–. Además la oferta de *frameworks* de aprendizaje automático existentes es muy amplia, con mucha variedad en nivel de abstracción. Hay muchos *frameworks* de

más bajo nivel, que implica una labor programática importante para construir la red de neuronas; pero existen también una gran variedad de *frameworks* de más alto nivel, que permiten elaborar los modelos de una forma más declarativa, sin necesidad de tanto código.

En general, `Python` tiene el equilibrio de un lenguaje rápido de usar y desarrollar, y la potencia de ser un lenguaje de propósito general, que combina varios paradigmas de programación. Esta combinación es muy beneficiosa para el desarrollo de experimentos o pequeñas pruebas de concepto que permitan validar modelos algo más complejos que los desarrollado solamente con lenguajes de *scripting*.

R [18] Es realmente un entorno libre de software completo –al estilo `Matlab`– sobre el que se construye un lenguaje de *scripting* pensado para trabajar con cálculos estadísticos. Huelga decir que desde su nacimiento ha evolucionado bastante, siendo actualmente un lenguaje muy usado en el ámbito de la ciencia de datos para, principalmente, protipados y análisis de datos.

Esta orientado a un uso matemático, lo que implica que cuenta con una gran variedad de funciones estadísticas y matemáticas Por ejemplo, cuenta con implementaciones propias de técnicas de clusterización o clasificación, y permite trabajar con funciones simbólicas. Además tiene librerías muy potentes para graficar *datasets*, así como unos *parsers* de ficheros muy potentes, que permiten transformar un csv a un objeto nativo de R de tipo *dataset* de forma inmediata. Estas funcionalidades y herramientas lo hacen muy interesante tanto para un análisis inicial de los datos como para las primeras aproximaciones de modelos.

A todo esto hay que añadirle que permite enlazar código en `C`, `C++` o incluso `Fortran` para la realización de las partes más pesadas o intensivas computacionalmente hablando. Incluso pueden manipularse desde programas en `C` los propios objetos de `R` –aunque hay que ser algo gurú para que esto salga bien–. Esta potencia casi permite implantar *scripts* de `R` en entornos productivos sin problemas.

Sin embargo, enfocando a *Deep Learning*, empieza a cojear. Existen muy pocas librerías de este ámbito en `R`. Tras la investigación iniciada, con suficiente uso y comunidad solo hay dos interesantes: MXNetR, la implementación de MXNet para `R`, que solo permite arquitecturas de redes de neuronas tradicionales y redes de neuronas convolucionales; y H2O, que permite redes de neuronas tradicionales y *deep autoencoders*.

Como propuesta de valor no es demasiado interesante ya que tanto MXNet como H2O tiene interfaz en otros lenguajes –entre ellos `Python` y `C++`, este último solo MXNet–. H2O ofrece algunas oportunidades llamativas en el sentido de que puede construir sus mode-

los sobre Spark o Hadoop, lo que resulta bastante interesante si se planteara un despliegue en un sistema real. Sin embargo, la potencia de H2O es propia del *framework* en sí mismo, no de la interfaz con R, que además no permite la elaboración de redes convolucionales, por lo que no es un valor añadido a R.

Elección final Finalmente la balanza se decantó por `Python`, que ha sido el lenguaje elegido. Uno de los motivos más importantes es la versatilidad del lenguaje frente a `R` y `C++`. Además, ha influido que mi conocimiento sobre `Python` –o `R`– era claramente superior al que tengo sobre `C++`. Eso, sumado a la lentitud de desarrollo general, ha provocado descartar `C++`.

Entre `R` y `Python` hay más discusión pues si bien `R` es un lenguaje más de *scripting*, está mucho más pensado y planteado para tareas estadísticas y matemáticas –como es la tarea que nos ocupa–. `Python`, por otra parte, ofrece las ventajas de tener un lenguaje multiparadigma con una gran cantidad de librerías disponibles. Esto, sin embargo, si bien parece interesante, no es demasiado útil para el problema que abordamos en este trabajo, pues muy posiblemente no usemos la mayor parte de esos recursos.

Ha sido, pues, una de las razones de mayor peso la cantidad de oferta *frameworks* enfocados al aprendizaje automático sobre los que nos permite trabajar `Python` lo que ha decantado la decisión a su favor. La cantidad de los mismos es claramente superior, contando en muchos casos con interfaces para *frameworks* implementado en otros lenguajes –`C++`, por ejemplo–, que sean mucho más eficientes y rápidos.

`Python` es, además, un lenguaje que está teniendo un gran impacto en los últimos tiempos en el ámbito del aprendizaje automático y la disciplina de Big Data. Se están desarrollando muchas herramientas sobre él –procesos ETL(extracción, transformación y carga), procesado de datos masivo, hasta Spark ha desarrollado una interfaz en `Python`–, y está bastante demandado por la industria actualmente.

3.2.2. Librerías

El estudio y elección del *framework* a utilizar está más o menos relacionada con la elección del lenguaje de programación que se haya escogido como lenguaje principal. En general, si bien en lenguaje no es tan importante, podía implicar ciertas restricciones sobre los *frameworks* a utilizar. De esta forma, aunque el análisis ha sido paralelo, era necesario cerrar uno para poder terminar de estudiar el otro.

La intención original era escoger un solo *framework* para unificar el código y reducir las dependencias de librerías de los experimentos. Como veremos a lo largo de esta sección, esa intención quedó finalmente descartada. Los *frameworks* estudiados que resultaban suficientemente interesantes, a priori, por el equilibrio entre abstracción y flexibilidad eran, aún así, demasiado exhaustivos programáticamente hablando. Esto dificultaba y encarecía –a nivel de esfuerzo– mucho los experimentos planteados, por lo que hubo que buscar alternativas.

A parte de las mencionadas a continuación, que han sido las más estudiadas, se barajaron algunas otras librerías descartadas más rápidamente:

- **Caffe**: Un *framework* muy potente implementado sobre C++ –que, de hecho, porta algunas de las mejores librerías de Matlab–. Sin embargo, lo descarté pronto debido a que para el uso de gran parte de sus funcionalidades con GPU requería ser escrito en C++. Además, aunque está especialmente orientado a clasificación de imágenes, el desarrollo sobre esta librería en ocasiones es muy farragoso.
- **CNTK**: A pesar de ser un *framework* con bastante buena reputación y API para Python –sobre C++–, tiene una licencia Open Source bastante restrictiva, lo que dificulta su uso y divulgación.

Sin embargo, sobre las librerías mostradas a continuación se realizó un estudio más intensivo al resultar más interesantes en la primera iteración del estudio:

TensorFlow TensorFlow [2] es un *framework* de aprendizaje automático desarrollado por Google y lanzado en noviembre de 2015. Es un *framework* que pretende ser de propósito general –dentro, obviamente, del aprendizaje automático–, más pensado casi como librería matemática que como *framework* abstracto. Se creó originalmente con el propósito de optimizar los procesos de aprendizaje automático y de *Deep Learning*. También se pretendía reemplazar a Theano en las tareas que desarrollaba Google.

Es un *framework* extremadamente flexible, puesto que permite –y obliga a– construir el grafo de operaciones que se pretenden ejecutar de forma totalmente personalizada. Una vez construido ese grafo se "compila" y se ejecuta con los datos de entrada que se faciliten. La generación de este grafo permite una paralelización muy grande, uno de los principales puntos fuertes de TensorFlow junto con su soporte de GPUs.

Esta flexibilidad obliga también a describir totalmente la red de neuronas y, en muchas ocasiones, a construir los procesos implicados –por ejemplo, describir exhaustivamente el proceso de *backpropagation* en redes de neuronas convencionales–. Esto hace que los desarrollos de los experimentos sean algo lentos si se desea que las redes sean algo más "convencionales" pero permitan modificar solamente ciertos parámetros concretos.

TensorFlow, sin embargo, es algo lento –sobre todo en comparación con otros *frameworks* escritos en C++ en su mayoría, como CNTK o MXNet–. Esto se debe a que todo el tratamiento del grafo computacional esta gestionado en Python –con sus consecuentes problemas de tiempo–. Además de que las operaciones con matrices no están totalmente pulidas, pues copian el contenido para operar, lo que dificulta mucho el procesado con matrices grandes, que suelen estar muy extendidas en este tipo de experimentos.

A pesar de esto, como podemos ver en el Cuadro 3.2, extraída de la tabla 7 de la bibliografía [14], si bien TensorFlow no es el *framework* más rápido de los estudiados, en los experimentos sobre muchos *cores* su mejora es muy significativa. Esto lo hace un actor a tener en cuenta, sobre todo en procesos que impliquen gran cantidad de datos susceptibles de ser paralelizados.

Cuadro 3.2: Comparativa entre *frameworks* (tiempo por minibach en segundos)

		Server CPU (Threads used)					
		1	2	4	8	16	32
FCN-R	CNTK	1.592	0.857	0.501	0.323	0.252	0.280
	TF	3.410	2.541	1.297	0.661	0.361	**0.325**
	MXNet	1.609	1.065	0.731	0.534	0.451	0.447
AlexNet-R	CNTK	9.381	6.078	4.984	4.765	6.256	6.199
	TF	6.124	4.229	2.200	1.396	1.036	**0.971**
	MXNet	17.994	17.128	16.764	16.471	17.471	17.770

Theano Theano [19] es una librería enfocada a Deep Leaning, cuyo uso esta muy extendido dentro de la comunidad. Está escrita íntegramente en Python, reimplementando alguna librería de este lenguaje –como Numpy, ya que usa unas estructuras de datos similares–.

Al igual que TensorFlow está pensada como una librería orientada a construir un grafo de operaciones que posterior se compila y se ejecuta. Sin embargo, cabe destacar

que los tiempos de compilación de este grafo son generalmente más lentos en Theano que en TensorFlow. Esto resulta algo llamativo pues los tiempos de ejecución son, en general, más lentos en TensorFlow. Además, Theano no permite paralelización en múltiples GPUs, algo que no afecta demasiado al trabajo actual –puesto que solo disponemos de una tarjeta gráfica– pero que conviene valorar.

También es relevante poner en valor –o precisamente lo contrario– que es una librería de "muy bajo nivel" dentro de las librerías de este tipo. Esto es especialmente sorprendente pues es una librería enfocada a *Deep Learning*, una disciplina lo suficiente concreta como para permitir cierto nivel de abstracción.

Sin embargo, a pesar de esto hay una gran cantidad de *frameworks* construidos sobre Theano –Keras o Lasagne, por ejemplo–, en su mayoría pretendiendo paliar algunas deficiencias como la abstracción o lo difícil que es trabajar con algunas de sus estructuras.

Keras Keras [3] es una librería enfocada a técnicas de *Deep Learning*, construida sobre Theano o TensorFlow. Es una librería muy potente que utiliza como motor Theano o TensorFlow pero proporcionando una gran capa de abstracción con una serie de funciones orientadas a construir de forma sencilla e intuitiva redes de neuronas utilizadas en *Deep Learning*.

Al usar Theano o TensorFlow por debajo puedes extraer la máxima potencia de cada uno de ellos –eligiéndolos según la ocasión lo requiera–, pero con la facilidad de desarrollo de una API de alto nivel. Además, es bastante flexible con la definición de las redes de neuronas, permitiendo definir la arquitectura de la red y la composición y parametrización de cada capa de forma libre.

Cabe destacar que tiene las mismas limitaciones de velocidad y rendimiento que Theano y TensorFlow –al estar construida sobre ellas– pero exprime al máximo sus capacidades en la construcción de redes de neuronas, por lo que no se añade un sobrecoste en rendimiento significativo. Sobre todo, más que optimizaciones sobre sus motores, ofrece un API mucho más claro y sencillo para construir concretamente redes de neuronas.

Naturalmente, limita parcialmente la potencia de sus *frameworks* base, sobre todo de TensorFlow, ya que uno de sus puntos fuertes era precisamente ser un *framework* más generalista. Sin embargo, por la naturaleza de nuestra tarea, para nosotros es extremadamente interesante esta abstracción.

Por último, cabe destacar que Keras permite utilizar una funcionalidad secundaria de TensorFlow con bastante utilidad, que es Tensorboard, una plataforma web que permite

analizar y estudiar los experimentos. Para ello genera de forma automática un resumen con las estadísticas más importantes y usadas del experimento para simplificar la tarea al científico.

H2O H2O [20] es más un entorno completo sobre el que construir soluciones de aprendizaje automático que un *framework* propiamente dicho. Combina en su motor una gran cantidad de técnicas –Map Reduce, compresión por columnas, múltiples modelos de aprendizaje automático– enfocadas a procesos de ETL y generación de modelos inteligentes. Parte de su potencia se debe a que este motor esta construido sobre Spark o Hadoop que, naturalmente, puede beber de múltiples fuentes de datos –SQL, NoSQL, S3, disco– y utiliza toda la potencia de HDFS –Hadoop Distributed File System, un sistema de ficheros optimizado para el tipo de operaciones que se suelen realizar sobre clústeres de Hadoop– para su procesado.

Como motor tiene una gran cantidad de modelos preimplementados, ofreciendo una capacidad de abstracción bastante importante, y proporcionando los modelos más usuales con los que se trabaja en la industria actualmente. Todo esto sobre tecnologías muy usadas en procesos de Big Data, algo muy jugoso para muchas empresas. Además, esos modelos pueden exportarse en varios lenguajes –R, Python, Scala y hasta como Java Objects– lo que permite utilizar la tecnología que mejor se adapte a las necesidades de cada tarea.

Sin embargo, tras estudiarlo más en detenimiento esa abstracción casi absoluta es precisamente su perdición, pues ofrece modelos muy potentes pero encorsetados. Para una labor más investigadora o experimentadora resulta acartonado y limitante, forzando a ciertos parámetros o arquitecturas que el *framework* tiene implementadas y optimizadas. Si es muy adecuado para entornos productivos, una vez probado y elaborado un modelo concreto, pues se adapta muy bien a las necesidades del mismo.

Elección final Finalmente la apuesta ha sido por la combinación TensorFlow con Keras. Toda la implementación se ha realizado en Keras, por su abstracción y facilidad de trabajo con el API, pero utilizando TensorFlow como motor. El uso de TensorFlow en lugar de Theano ha venido motivado principalmente por la mayor optimización en el uso de GPUs del primero frente al segundo, así como la potencia que ofrece TensorFlow en los procesados paralelos.

En general, como se comenta anteriormente, los tiempos de ejecución de Theano son más rapidos que los de TensorFlow. Esto puede parecer decantar la balanza hacia Theano, pero en la literatura se pueden encontrar múltiples comparativas que dan a TensorFlow ven-

taja en algunas conclusiones muy concretas. Particularmente, se ha detectado que Tensorflow bate los tiempos de Theano con CPUs en redes grandes o con muchos datos [21]. Esto es interesante pues la experimentación en CPU es más sencilla que en GPU. No obstante, la mayor parte de la experimentación de este trabajo se realiza en GPU, y se ha podido encontrar que TensorFlow es especialmente rápido en gráficas de NVIDIA GTX 1080 [22] construido sobre Keras, en comparación con Theano sobre Keras. Esto resulta muy interesante pues ya se plantea utilizar una combinación de librerías de Theano o TensorFlow –como motor– y Keras –como librería de implementación– debido a su abstracción.

El mayor problema encontrado hasta el momento, que es la dificultad para elaborar redes optimizadas de forma sencilla, se soluciona con el uso de Keras, mientras que el motor de TensorFlow aporta características muy similares a las de Theano con alguna ventaja para nuestro caso actual.

H2O, por otra parte, quedó descartado tras explorar algo más su documentación por los motivos mencionados anteriormente: no permite la flexibilidad necesaria para la parametrización que se plantea en este trabajo. Ha sido una oportunidad interesante de explorar un *framework* bastante pulido, pero que no se adecua a las necesidades del proyecto.

4. Instalación del entorno

Durante la instalación del entorno en el que se iban a desarrollar los experimentos fui consciente de la poca documentación que hay, en muchas ocasiones, sobre este paso. La instalación de los drivers de NVIDIA en Linux –en este caso concreto en sistemas basados en Debian– no está demasiado documentado y el proceso completo no se encuentra en ningún sitio. De esta forma, he decidido dedicar unas páginas a documentar los pasos seguidos para la correcta instalación de NVIDIA, CUDA y cuDNN en un `Ubuntu 16.10`.

Al final de esta sección el lector podrá tener configurado y funcionando una instalación limpia de TensorFlow y Keras sobre NVIDIA y CUDA 8. Se asume que se parte de un equipo instalado con Ubuntu 16.10 y python3, que disponga de una gráfica NVIDIA.

A lo largo de esta sección, en los *snippets* de código se representará el código en `bash` con el símbolo `$` para representar el *prompt*.

4.1. Instalación de NVIDIA, CUDA y cuDNN

Este proceso se estructura en tres pasos claramente diferenciados: la instalación del driver de NVIDIA, la instalación de CUDA y la instalación de la librería cuDNN.

4.1.1. Instalación de NVIDIA

Este es el paso más delicado de toda la instalación, ya que de ir mal probablemente impida el arranque de la sesión gráfica de Ubuntu, lo que puede ser bastante molesto. Existen dos formas de instalar los drivers de NVIDIA: desde el repositorio de paquetes de Ubuntu, o desde la propia página de NVIDIA. La opción que recomiendo, por comodidad y sencillez es desde los repositorios del sistema operativo, aunque si se desea explotar al máximo la gráfica probablemente fuese más interesante instalar los drivers privativos directamente desde la web de NVIDIA.

En esta guía nos centraremos en la instalación desde el repositorio, pero incluyo alguna aclaración para los aventureros que escojan el software privativo. NVIDIA proporciona un *runtime* que se ejecuta como un *script*. Es especialmente importante si se usa esta opción descargar también la firma que proporciona NVIDIA para el driver, que debe ser introducida en el arranque de la `BIOS` si esta es `EFI`. De otro modo, la `BIOS` no reconocerá la firma del

driver –ya que no tendría– y no permitiría la ejecución del mismo. Si se firma correctamente el driver no debería haber mayor complicación, pero para realizar este proceso no incluyo información en esta guía.

Para la instalación desde el gestor de paquetes, el proceso es bastante más sencillo. Basta con buscar cuál es la última versión –se puede usar el tabulador tras introducir en la terminal `sudo apt-get install nvidia-` para ver cuales son las opciones– e instalar esa:

```
$ sudo apt-get update
$ sudo apt-get install nvidia-375
```

Tras esto debe reiniciarse el sistema y ya tenemos instalado correctamente el driver de NVIDIA. Para comprobarlo podemos ejecutar la sentencia:

```
$ nvidia-smi
```

Esta sentencia nos ofrece información del estado de carga de la gráfica –de forma similar a `top` con la CPU–.

4.1.2. Instalación de CUDA

Una vez instalado correctamente el driver de NVIDIA, podemos iniciar la instalación de CUDA. Para ello primero debemos descargar el software de instalación de CUDA, que se puede encontrar en https://developer.nvidia.com/cuda-downloads. Seleccionando la arquitectura de nuestro Linux y la opción *runtime*, se nos descargará un *script* en `bash`.

Al igual que en el paso anterior indicaba que era mucho más interesante instalar el driver de NVIDIA desde el gestor de paquetes para facilitar la instalación y la actualización, con el driver de CUDA es preferible instalarlo mediante su *script*. De otra forma, la localización que escoge el sistema operativo no suele estar reconocida por la mayoría de *frameworks* que trabajan sobre CUDA y genera problemas.

El proceso de instalación es relativamente sencillo. Debe ejecutarse:

```
$ sudo sh <cuda runtime> --override
```

Una vez ejecutado, es muy importante asegurarse de que se marcan las siguientes opciones:

- No deseamos instalar el driver de NVIDIA, ya que lo hemos instalado previamente. Es Extremadamente importante no instalar el driver de NVIDIA en este paso ya que entrará en conflicto con el previamente instalado y dejará el sistema operativo inestable.
- La localización de instalación debe ser la de por defecto, `/usr/local/cuda`
- La instalación de ejemplos es opcional, aunque pueden servir para probar la instalación.

Tras la ejecución del *script*, se nos habrá generado una carpeta en la localización escogida `/usr/local/cuda<version de CUDA>`. Personalmente recomiendo realizar un enlace simbólico de ese directorio al mismo sin la versión `/usr/local/cuda`, que es el que buscarán la mayoría de los *frameworks*. En cualquier caso, tras la instalación de CUDA, para un correcto funcionamiento deben ejecutarse las siguientes lineas:

```
$ export PATH=/usr/local/cuda/bin:$PATH
$ export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
```

Este último paso es muy importante ya que son variables que algunos *framework* –TensorFlow entre ellos– buscan para localizar los binarios de CUDA. Estas dos sentencias deberían incluirse en el fichero `.bashrc` para que se ejecuten al inicio de cada sesión.

4.1.3. Instalación de cuDNN

Tras los pasos anteriores podemos iniciar la instalación de las librerías de cuDNN que es –con diferencia– la más sencilla. Esencialmente consiste en descargar unos ficheros y copiarlos a la localización de la instalación de CUDA. Hay una pequeña salvedad, debemos entrar y registrarnos en https://developer.nvidia.com/cudnn, la web de cuDNN. Tras el registro podremos descargar la librería. Es muy importante que al descargarla escojamos la opción *cuDNN v7.0 Library for Linux* dentro de la opción para CUDA 8.0 –que es la que instalamos previamente–. Esto se debe a que la otra opción para Linux –un `.deb`– en ocasiones da problemas al instalarse ya que no lo hace en el lugar correcto.

Esto nos descargará un fichero `.tgz` que debemos descomprimir y mover a la raíz de nuestro directorio de instalación de CUDA. Realmente la librería trae dos carpetas –`include` y `lib64`– que también se encuentran en nuestra instalación de CUDA y que son las que deben fusionarse, para incluir el código de cuDNN en la instalación de CUDA.

Una vez incluidos estos ficheros en la instalación de CUDA ya tenemos una instalación completa de NVIDIA, CUDA y cuDNN. Estamos listos tanto para instalar nuestros *frameworks* como para usarlas con otros objetivos.

4.2. Instalación de TensorFlow y Keras

Una vez completado el proceso de instalación del sistema operativo y de los drivers y librerías de NVIDIA correcta estamos en condiciones de proceder con la instalación de TensorFlow y Keras. La instalación en sí mismo de ambos es relativamente sencilla, pero hay alguna particularidad que quiero hacer notar.

Al ser Keras una librería enfocada a las redes neuronales que provee una API sobre otros motores, es necesario instalar previamente dichos motores. Como ya se ha visto anteriormente, en el caso particular de Keras, estos motores son TensorFlow [2], Theano [19] o CNTK [23].

Además, ambas librerías pueden ser instaladas mediante un gestor de paquetes –bien del sistema, como en el caso de TensorFlow, o del lenguaje, como pip, en caso de ambas librerías–, o desde los ficheros fuente. En este documento se indicará el proceso para hacerlo mediante el gestor de paquetes del lenguaje –pip– ya que facilita la actualización y el mantenimiento de la librería. Sin embargo, de querer instalarse desde los ficheros fuente, en la web de cada una de las librerías se incluyen indicaciones para llevar a cabo el proceso.

Al instalarlo desde el gestor de paquetes de `python`, podemos instalar las librerías directamente en el sistema operativo o –como recomiendo y haré yo– instalarlas en un entorno virtual, de forma que no interfiera con las instalaciones previas realizadas en el sistema operativo.

Por último, pese a que estas indicaciones están redactadas por mi en base al proceso que seguí durante la instalación del entorno, en este caso se corresponden totalmente con los pasos que se pueden encontrar en las propias páginas web de TensorFlow y de Keras. El motivo por el que las he incluido a continuación es por claridad y con la intención de agrupar todo el proceso en un único lugar.

Como ya he mencionado anteriormente, el primer paso es instalar la librería "core" o la que hará como motor de Keras, en este caso TensorFlow. Naturalmente este *framework* funciona de forma independiente y puede ser usado de forma individual. El primer paso sería la instalación –en caso de no haberlo hecho ya– del paquete para crear entornos virtuales y

la propia creación del mismo:

```
$ apt-get install python3-dev python3-pip
$ pip3 install virtualenv
$ virtualenv -p python3 <localizacion del virtualenv>
```

Esto nos creará en la localización elegida un entorno virtual con `python3`. Otra ventaja que ofrece este proceso, además de tener nuestras librerías independizadas de las base del sistema operativo es que podemos especificar –como en este caso– que solamente se instale `python3` y se use como versión por defecto de `python`, simplificando muchos problemas de compatibilidades y versionados.

A continuación debemos activar el entorno virtual e instalar la librería de de TensorFlow. En este paso es muy importante tener activado el entorno virtual y haberlo creado con la opción de `python3`, ya que de otra forma se instalará la versión de TensorFlow de `python2`.

```
$ source <localizacion del virtualenv>/bin/source
$(virtualenv) pip install --upgrade tensorflow-gpu
```

Es importante que en la terminal veamos el nombre del entorno virtual –el nombre de la carpeta donde se encuentra– entre parentesis. Esto indica que el entorno virtual se ha cargado correctamente. Este paso es tan importante porque al ejecutar `pip` dentro del entorno, automáticamente escogerá la versión de `python3` (que es la que hemos instalado). Tras estos pasos ya tenemos correctamente instalado TensorFlow con su versión más reciente, preparado para el uso en GPUs. Es interesante tras este paso validar la instalación, y aunque para no alargar este documento no se incluirá aquí, se puede encontrar en la documentación oficial [24].

A continuación estamos listos para instalar correctamente Keras y empezar a usarlo. El proceso de instalación de Keras es aún más sencillo –en parte debido a que ya hemos instalado `virtualenv`–. Sin salir del entorno virtual debemos ejecutar:

```
$(virtualenv) pip --upgrade install keras
```

Tras estas lineas, ya tenemos instalado completamente en nuestro sistema los dos *frameworks* usados en este trabajo así como los drivers de una tarjeta gráfica NVIDIA y las librerías necesarias para computar redes de neuronas sobre ellas.

Como se venía viendo en secciones anteriores, una de las ventaja de utilizar Keras

para la construcción de las redes es que el mismo código puede ejecutarse utilizando otros motores –Theano o CNTK– solamente cambiando el *framework* que se instala como motor.

5. Experimentación y estudio

En esta sección se desarrolla el grueso principal del trabajo. Se han implementado dos tipos de arquitectura –MLP y CNN–, que se estudiarán por separado para analizar sus componentes principales. Cada una de esas arquitecturas se ha aplicado a los dos dominios estudiados de forma similar –es decir, con los mismos parámetros–. Naturalmente, las entradas y salidas se han debido variar en cada problema, pero la estructura general se ha mantenido para poder comparar en igualdad de condiciones.

En la última parte de esta sección se comentará el estudio comparativo entre ambas arquitecturas para cada problema, analizándolo en profundidad. De igual forma, en la sección siguiente pueden encontrarse las conclusiones generales –tanto de la experimentación y análisis como del trabajo en su conjunto–, tratando de responder a los objetivos planteados.

Los resultados de la experimentación se muestran, en general, con forma de gráfica (precisión del conjunto de entrenamiento frente al de test) o con forma de cuadro que recoge los resultados al final del entrenamiento. El conjunto de ambos es suficientemente claro en la mayor parte de los casos como para considerarse adecuado para el estudio. En la bibliografía, de hecho, normalmente se suele utilizar como medida exclusivamente la precisión de la red en base a los parámetros utilizados, pero hay ciertos comportamientos que han sido más sencillos de mostrar por medio de gráficas.

5.1. MLP

Para el estudio del Perceptrón Multicapa –MLP– se ha escogido una arquitectura tradicional. Si bien se realizó un estudio inicial sobre el interés de probar arquitecturas con múltiples capas, en aras de la síntesis y de realizar un análisis concreto se descartó finalmente. En base al teorema sobre la capacidad de aproximación universal [25] en MLPs con una sola capa, finalmente la arquitectura escogida ha sido de una sola capa oculta puesto que puede aproximar cualquier función.

Esta arquitectura, como puede observarse en la Figura 5.1 es relativamente sencilla. Está compuesta por tres capas: la capa de entradas, la capa oculta y la capa de salida. La capa de entrada simplemente recibe la imagen y tiene tantas neuronas como pixels tenga la imagen. En el caso del MNIST son 784 y en el caso del CIFAR son 3072 –tengamos en cuenta que el CIFAR tiene tres canales de color–. El número de neuronas de la capa oculta

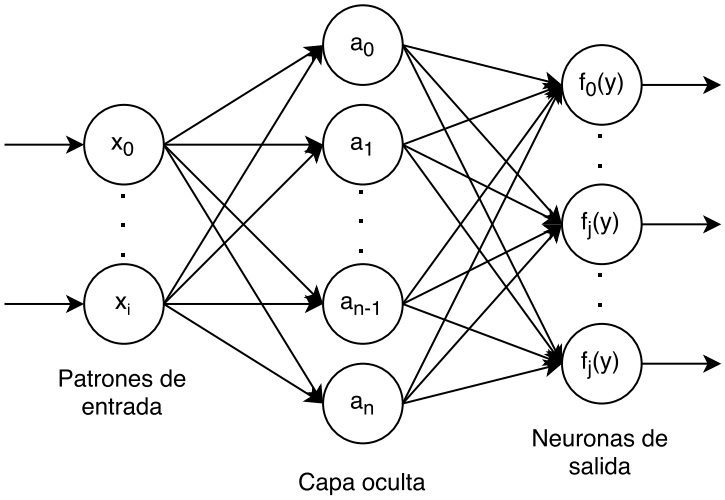


Figura 5.1: Arquitectura del perceptrón multicapa

ha ido variando con los experimentos, pues es uno de los parámetros estudiados. La cantidad de neuronas de salida, por el contrario, ha sido igual en ambas redes –diez– pues los dos conjuntos de datos se clasifican en diez clases –del 0 al 9 en el MNIST, 10 clases diferentes en el CIFAR.

Las neuronas de la red están conectadas completamente –esto es que cada neurona esta conectada con todas las neuronas de la capa siguiente–, la función de activación de la capa oculta ha sido generalmente sigmoide –aunque se ha cambiado a una ReLU para estudiar su impacto–, y la función de activación de la capa de salida se correspondía con una función softmax. La función softmax –que se puede ver a continuación– se utiliza con mucha frecuencia como función de activación de la capa final en tareas de clasificación. Esto se debe a que es una función que, esencialmente, recibe un vector de valores reales y lo transforma en un vector de valores reales entre 0 y 1 cuya suma es 1. Esto es especialmente interesante en las tareas de clasificación pues transforma las salidas de la red a una aproximación probabilística –entre cero y uno–, que son también valores más entendibles por el ser humano. Con una función de activación softmax se puede generar una función de error de tipo cross-entropy, mucho más adecuada para tareas de clasificación. Junto con esta función se ha utilizado también descenso de gradiente estocástico, la función de optimización habitual de los MLPs.

$$f_j(y) = \frac{e^{y_j}}{\sum_k^y e^{y_k}}$$

Función softmax, donde y se corresponde con el vector –en nuestro caso las salidas de las neuronas

de la red–, j hace referencia a la neurona en la que se aplica la función softmax y k representa la posición del sumatorio que va desde cero hasta la cantidad de neuronas. Fuente: http://cs231n.github.io/linear-classify/#softmax

5.1.1. Análisis de parámetros

Debe tenerse en cuenta que el MNIST es un conjunto de datos interesante para su estudio –por la amplia cantidad de estudios previos y su simplicidad de tratamiento–, pero ver el impacto de los diferentes parámetros puede resultar algo arduo. Esto se debe a que el margen de mejora es relativamente pequeño, por lo que se tendrá que prestar mucha atención. Esencialmente se ha estudiado el impacto del ratio de aprendizaje, de la cantidad de neuronas, y de la función de activación.

Se ha barajado ampliamente introducir un breve estudio sobre el número de iteraciones, pero finalmente no se ha considerado demasiado interesante. Por los experimentos realizados se ha visto que el impacto de las iteraciones se nota solamente a efectos de tiempo de entrenamiento. Es decir, el CIFAR es un conjunto que requiere, de media, más iteraciones que el MNIST para alcanzar una estabilidad, pero la cantidad de iteraciones no es relevante para el desarrollo de la red. Normalmente si se deja una gran cantidad de iteraciones sí se ha visto que se producen pequeñas mejoras –en el MNIST, sobre todo–, pero muchas veces no merecen la pena el esfuerzo.

Sí se ha notado, sin embargo, que la facilidad para caer en mínimos locales es mayor en el CIFAR que en el MNIST. El MNIST si se deja tiempo suficiente sigue mejorando, aunque de forma muy lenta; sin embargo, el CIFAR llega más rápidamente –relativa a la cantidad de iteraciones necesarias para estabilizarse– a una estabilidad tal que no se produce mejora.

Algo que ha resultado en general interesante es que no se han detectado signos de sobreajuste –aunque era algo que ya se esperaba–. Sí se han visto experimentos en los que los conjuntos de test y de entrenamiento han divergido terriblemente, estancándose a niveles de precisión muy diferentes –como puede verse en la Figura 5.2, pero nunca el conjunto de test ha empezado a perder precisión drásticamente.

Ratio de aprendizaje El ratio de aprendizaje es un parámetro que afecta al descenso de gradiente estocástico. Este descenso de gradiente recordemos que es usado para el cálculo del aprendizaje de la red, de forma que se optimice el error para reducirlo al mínimo. El

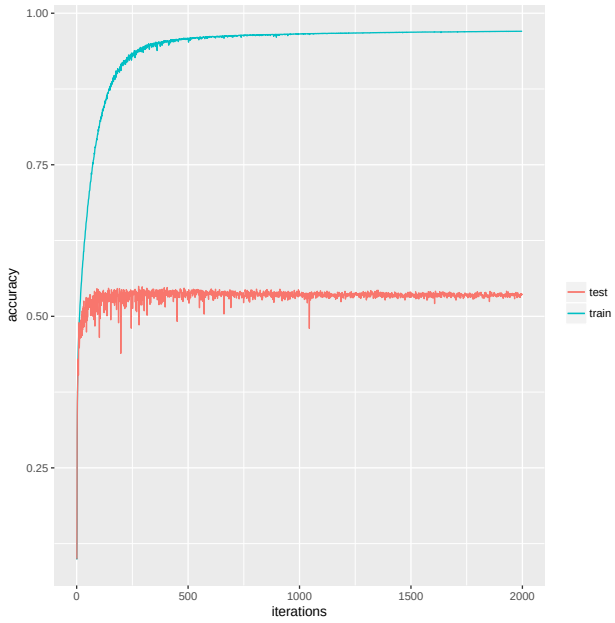
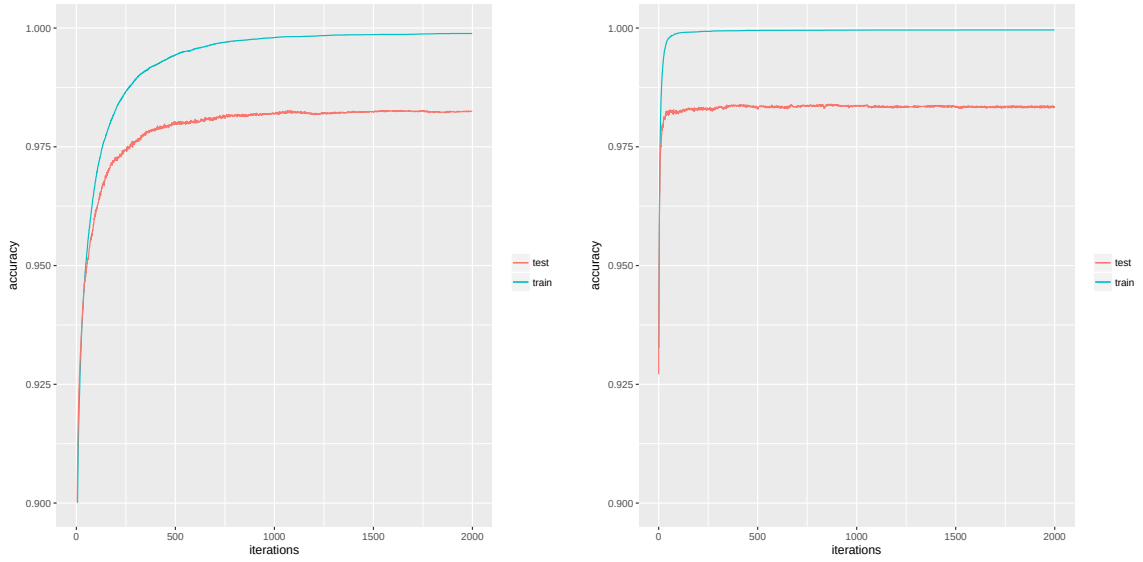


Figura 5.2: Brecha entre la precisión de entrenamiento y test en CIFAR

ratio de aprendizaje regula la velocidad con la que converge el gradiente, por decirlo de otra manera, la dimensión del "salto" que da el error. Está muy ligado a la capacidad de la red para evitar mínimos locales, pues con un ratio de aprendizaje alto el "salto" es mayor y es más difícil que se detenga en un mínimo local. Generalmente con un ratio de aprendizaje alto se producen oscilaciones más grandes de la función de error respecto al mínimo, mientras que con ratios de aprendizaje más bajos la convergencia suele ser más lenta y es más fácil que el aprendizaje encalle en mínimos locales.

El estudio del ratio de aprendizaje ha llevado por caminos extraños e interesantes. El primer hecho sorprendente ha sido la aparente correlación –o al menos, relación– entre el ratio de aprendizaje y el tamaño del batch. No se han incluido experimentos respecto a este punto pues no ha sido algo especialmente estudiado, pero si ha resultado llamativo. Con un tamaño de batch mayor se puede –y debe, de hecho, para obtener buenos resultados– agrandar el ratio de aprendizaje hasta tamaños sorprendentemente altos (como 0.8, 2 o incluso 4) sin que el aprendizaje sufra un impacto importante. La red tiene, de hecho, una convergencia mucho más rápida en caso de utilizar *minibatch* grandes con ratios de aprendizaje altos, aunque se penaliza la precisión.

Como se puede ver Figura 5.3b –comprobando con el Cuadro 5.1–, el impacto del *learning rate* si es notable. Recordemos que en el MNIST una diferencia de resultado de



(a) Ratio de aprendizaje de 0.1

(b) Ratio de aprendizaje de 2

Figura 5.3: Diferencia entre MLP con 900 neuronas entrenado durante 2000 iteraciones en MNIST, variando el ratio de aprendizaje

0,01 es una variación notable dados los resultados que suelen obtenerse. En general las diferencias son mínimas, pero sí se puede comprobar que con un ratio de aprendizaje más alto se obtienen convergencias mucho más inmediatas y, en muchas ocasiones, algo más de precisión. Naturalmente debe encontrarse un equilibrio, pues con un ratio muy alto el aprendizaje entre iteraciones se vuelve muy inestable –la precisión oscila–, teniendo valores de precisión muy dispares.

Hay una excepción a esta norma –que con ratios de aprendizaje altos la red oscile mucho–, que se da al utilizar funciones ReLU. Al usar este tipo de funciones los ratios de aprendizaje funcionan mejor. Esto se debe a que este tipo de función –lineal rectificada– prolonga, pero también atenúa, la distancia a la que se mantiene el error propagado. Por eso, con ratios más altos tiene más peso el error calculado en las últimas iteraciones, lo que mejora este algoritmo.

Número de neuronas El número y distribución de neuronas en la arquitectura de una red de neuronas es, probablemente, uno de los parámetros más estudiados de las redes. Y también uno de los parámetros que más depende del problema concreto que se esté abordando. En el estudio de este parámetros es absurdo hacer ningún tipo de comparativa entre CIFAR y

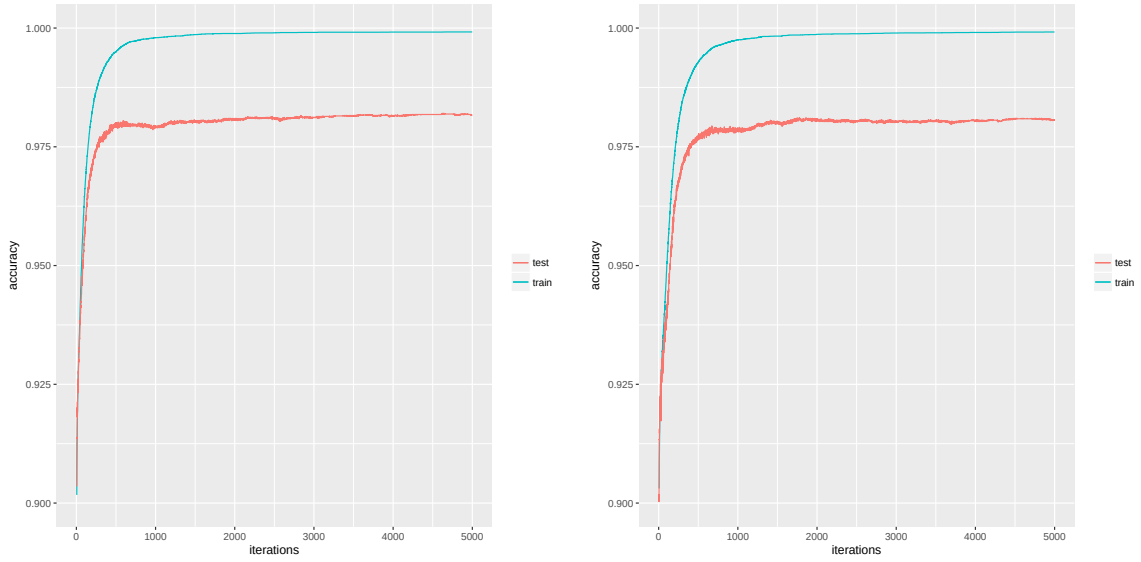
Cuadro 5.1: Comparativa entre el ratio de aprendizaje de diferentes configuraciones de redes MLP

Conjunto de datos	Número neuronas	Ratio de aprendizaje	Precisión en entrenamiento	Precisión en test
MNIST	900	0.1	0.994	0.977
		0.4	0.999	0.982
		2	0.994	0.978
	500	0.1	0.995	0.978
		0.4	0.999	0.981
		2	0.995	0.978
CIFAR	900	0.1	0.971	0.546
		0.4	0.979	0.549
		2	0.970	0.546
	500	0.1	0.425	0.415
		0.4	0.527	0.482

MNIST, pues la elección del número de neuronas va a venir dado por MNIST y CIFAR de forma independiente.

A parte del hecho de que la elección del número de neuronas es muy dependiente del problema, como podemos observar en la Figura 5.4, el impacto de las mismas no es demasiado importante. La precisión y el comportamiento del MLP son prácticamente similares a no ser que variemos significativamente el número de neuronas (del orden de 3000 o 5000). Hay que tener en cuenta, de igual forma, que tanto el CIFAR como el MNIST tienen una cantidad de variables de entrada relativamente alta –3000 y pico el primero, casi 800 el segundo–, por lo que las variaciones de neuronas deben hacerse en ese orden de magnitud para que tengan algún impacto.

Aún así, una vez escogido un número de neuronas que funciones bien, el resto de parámetros tienen un impacto mucho mayor que variar el número de las neuronas. Incluso se han realizado algunas pruebas iniciales con varias capas para comprobar si era o no interesante su estudio, y los resultado fueron bastante decepcionantes. Con dos capas de 300 y 200 neuronas los resultados han sido incluso peores que con una única capa de 500 neuronas, lo cual era esperable debido al problema de *Vanishing Gradient*. Este problema está muy relacionado con el funcionamiento del descenso de gradiente, pues al actualizar los



(a) 500 neuronas (b) 1500 neuronas

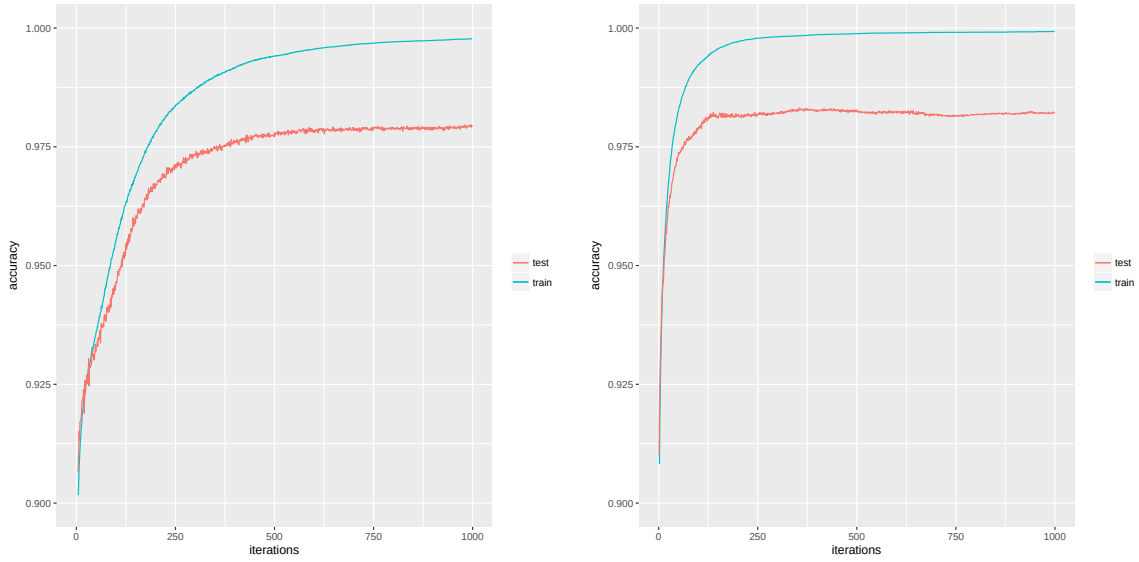
Figura 5.4: Variación del número de neuronas en un MLP entrenado durante 5000 iteraciones en MNIST

pesos con el error que proviene de la capa anterior, se "'pierde" más error con dos capas que con una.

Función de activación El estudio de la función de activación ha sido uno de los más sorprendentes e interesantes de los realizados. En primera instancia no parecía demasiado importante la función de activación de la red más allá de que la mayoría de la literatura utilizaba sigmoides o tangenciales –hiperbólica o arcotangente–. Sin embargo, dada la poca información respecto al por qué de esa elección motivó algunos experimentos al respecto.

Si bien en la Figura 5.5 se puede ver la evolución del entrenamiento de una red con activación sigmoidal –Figura 5.5a– frente a una activación ReLU –Figura 5.5b–, es más sorprende la Figura 5.6. En la Figura 5.5b se puede advertir uno de los primeros impactos de la ReLU, se genera mucha más divergencia entre el entrenamiento y el test, de forma que la red se aprende sobre el conjunto de entrenamiento más que al utilizar una función sigmoide. Atendiendo a las precisiones, con la función sigmoide obtenemos una precisión de 0,957 en entrenamiento y 0,947 en test; mientras que con la función ReLU tenemos un 0,999 en entrenamiento y un 0,982 en test.

Como se puede ver, el entrenamiento con las ReLU obtiene resultados algo mejores,



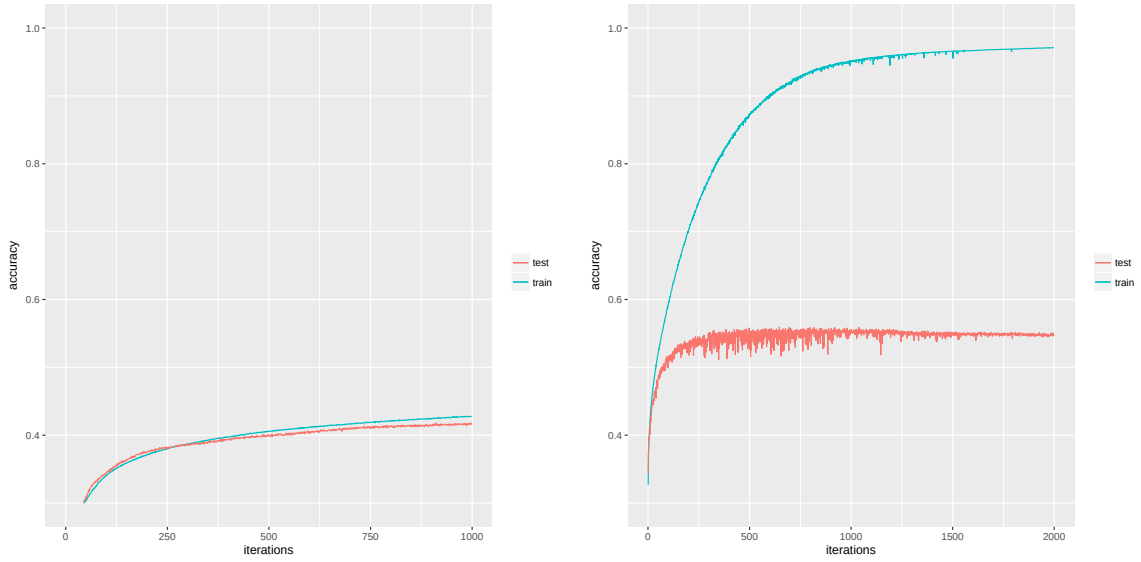
(a) Función de activación sigmoide (b) Función de activación ReLU

Figura 5.5: MLP con 900 neuronas ocultas entrenado durante 1000 iteraciones con ratio de aprendizaje 0.4 en MNIST

además de tener una convergencia más rápida. Y se ve que la convergencia es más rápida con funciones ReLU. Que estabilice antes tampoco es necesariamente mejor pues está muchas iteraciones con un crecimiento casi constante, necesitando muchas iteraciones para mejorar la precisión. Por otro lado, con funciones sigmoidales se obtienen resultados razonablemente buenos y el crecimiento es algo más lineal. Hay que tener en cuenta un detalle difícil de mostrar en este tipo de gráficas, y es que los tiempos de ejecución de cada iteración son mayores con la función ReLU –apenas unos segundos por iteración –de 15 segundos con sigmoidales a 20 con ReLU–, pero puede ser notable en ejecuciones largas–.

En la Figura 5.6b se ve aún más dramáticamente la brecha que sufre la red. Nótese que aunque esa red aún tiene margen de mejora antes de estabilizar –en entrenamiento–, ya se puede ver una diferencia notable entre la precisión en entrenamiento y la precisión en en test: de casi 0,5 puntos sobre 1. Además, como ya veníamos anticipando al hablar del ratio de aprendizaje, se puede ver como al utilizar funciones de activación ReLU, un ratio de aprendizaje alto impacta muy negativamente en el aprendizaje de la red, haciendo que tenga mucha variabilidad entre los resultados de cada iteración. Como se ve en la figura, pese a haber estabilizado en algo más de 0,5, la oscilación de resultados es de casi 0,03 puntos.

Esta brecha se debe principalmente a la función ReLU. Por la forma en la que



(a) Función de activación sigmoide (b) Función de activación ReLU

Figura 5.6: Diferencia de la función de activación en un MLP con 900 neuronas ocultas entrenado durante 1000 iteraciones con ratio de aprendizaje 0.1 en CIFAR

se actualizan los pesos –en base a la derivada del error, que viene dado por la función de activación–, esta actualización es mucho más efectiva. Eso quiere decir que con funciones ReLU, la capacidad de mejora en entrenamiento aumenta. Naturalmente, no es entrenamiento efectivo pues esa mejora no se refleja en el conjunto de test.

Una detalle que es ligeramente sorprendente es que, atendiendo a la Figura 5.6b, la red sufre un pequeño sobreajuste. Si se presta atención, a partir de la iteración 1000 la tendencia del conjunto de test desciende un poco, aunque se compensa con el descenso drástico de la oscilación.

5.2. CNN

Para el estudio de la red de neuronas convolucional se ha utilizado, al igual que con el MLP, una arquitectura "canónica". Esta arquitectura, como se ve en la Figura 5.7, esta compuesta por dos capas convolucionales, dos capas densas –totalmente conectadas– y la capa de salida. Cabe desatacar que cada capa convolucional consta realmente de dos capas, que se han agrupado en una al esquematizarse por una cuestión de síntesis. Esas dos capas son la capa convolucional propiamente dicha, que en este caso es una convolución en dos

dimensiones con función de activación ReLU y 32 filtros con un kernel de $3x3$; y una capa de agrupación –*pooling*–, en este caso un *maxpooling* de $2x2$, es decir, que de cada conjunto de $2x2$ píxeles resulta el máximo. La segunda capa convolucional de la figura tiene la misma estructura pero con capas convolucionales de 64 filtros. Se ha añadido a los conjuntos de datos un pequeño proceso de *data augmentation*, que es un proceso para ampliar el conjunto de datos mediante transformaciones a las imágenes. En este caso simplemente se han aplicado desplazamientos verticales y laterales, lo que no es especialmente significativo.

Las capas densas tienen el mismo número de neuronas, que varía con los experimentos, y una función de activación ReLU, mientras que la capa de salida tiene diez neuronas –una por clase, en ambos conjuntos– y una función de activación softmax. En este caso la función de error utilizada es de tipo *categorical crossentropy*, mientras que la función de optimización es de tipo RMSprop con un decaimiento de 10^{-6}.

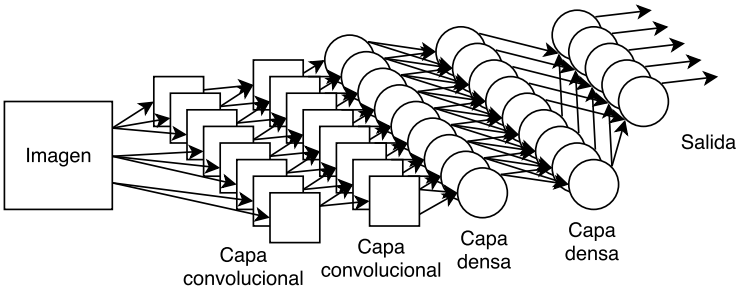


Figura 5.7: Arquitectura de la red de neuronas convolucional

Es importante comentar tres cuestiones de lo mencionado anteriormente: el funcionamiento –al menos de forma intuitiva– de las capas convolucionales, el funcionamiento de las capas de *maxpooling* y el funcionamiento de RMSprop. El funcionamiento de las capas convolucionales viene supeditado a una premisa que se asume al utilizar redes CNN, que es que la entrada va a ser una imagen –o va a tener forma de ella–. Esto es que será una matriz n-dimensional –una dimensión por cada canal de color– y rectangular. Las capas convolucionales lo que hacen es definir n filtros –como si fueran neuronas– que tienen un tamaño de kernel, que esencialmente es el tamaño del trozo de imagen que van a procesar, y se "desplaza" la convolución por la imagen. De esta forma, el primer filtro de una capa con kernel de $4x4$ abarcará de la posición 0 a la 3 de la imagen. Si el desplazamiento definido es 1 –como en nuestro caso– el segundo filtro abarcará de la posición 1 a la 4. Y de esta forma tantos filtros como se tenga. Esto permite que la capa convolucional se especialice en reconocer características concretas –cada filtro una–.

La capa de *pooling*, por su parte, que tiene el mismo número de nodos que la capa convolucional –uno por cada neurona convolucional– lo que hace es procesar la mini-imagen resultante del filtro reduciendo su tamaño n veces, siendo n el tamaño del *pooling*. En nuestro caso, por ejemplo, al tener un *pooling* de $2x2$, reduciremos el tamaño de la imagen a la mitad. Las políticas de *pooling* pueden ser desde el máximo hasta el mínimo, pasando por medias ponderadas o la medida que nos parezca, aunque típicamente se usa el máximo.

Por último, la función de optimización de RMSprop es una función construida sobre el descenso de gradiente de forma que realiza una adaptación del ratio de aprendizaje con el paso del tiempo. Se divide el *learning rate* en cada peso por la media de los últimos resultados del gradiente –calculados de esta misma forma y atenuados por el parámetro de decaimiento– para ese peso. Es una forma de descenso de gradiente muy usado en CNNs, sobre todo al utilizar *minibatch*, pues pesa menos el impacto del resto de pesos y mucho más el de resultados previos del propio peso. Además, puede modificarse el decaimiento en conjunción con el *learning rate* para controlar el peso del mismo de forma más profunda.

5.2.1. Análisis de parámetros

Durante el análisis de los parámetros en las CNNs se han estudiado prácticamente los mismos para dar coherencia al estudio general. Sin embargo, a lo largo del análisis se tratarán algunos parámetros más, asociados particularmente a este tipo de redes. Estos otros parámetros se han considerado interesantes también y por eso se han incluido.

En esta arquitectura tampoco se ha visto relevante un análisis del número de iteraciones, aunque cabe destacar que se han visto reducidos los tiempos. Concretamente, en torno a las 500 o las 1000 iteraciones ya habían convergido las redes en ambos conjuntos. Esto es algo que contrasta bastante con las arquitecturas tradicionales, en las que muchas veces se necesitaba el doble de iteraciones para lograr estabilidad.

Al igual que en el estudio sobre MLPs resultaba sorprendente la brecha entre entrenamiento y test en el conjunto CIFAR, en este estudio se han encontrado otras cuestiones generales interesantes. Particularmente, como puede verse en la Figura 5.8, se ha detectado cierto grado de sobreaprendizaje más drástico que en el dominio anterior.

Se puede apreciar bastante oscilación en la precisión de entrenamiento, que se comentará en el apartado del ratio de aprendizaje, pero sobre todo es sorprendente el descenso de precisión que sufre el conjunto de test. Este tipo de sobreaprendizaje es chocante porque no se había visto anteriormente con los MLPs. En ambos conjuntos de datos, las redes podían

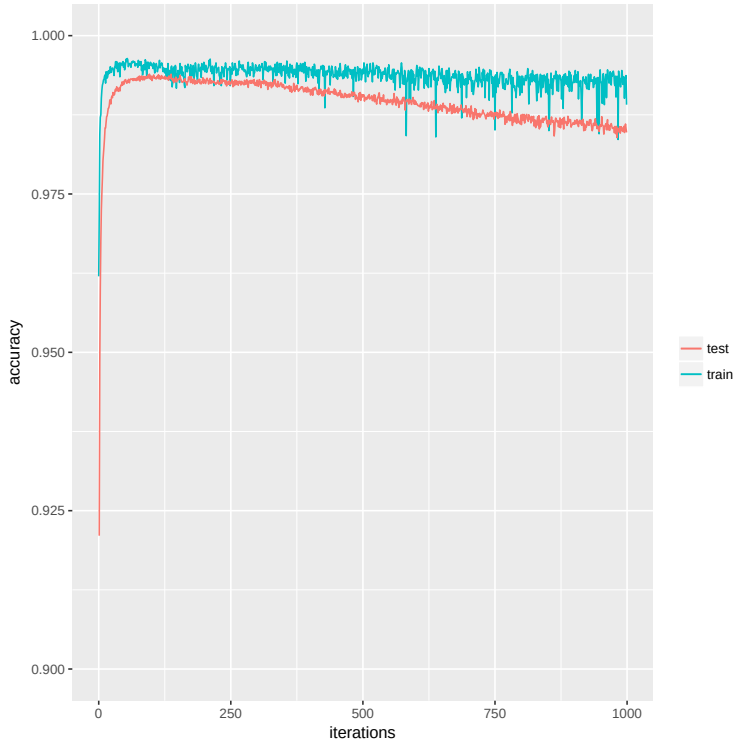


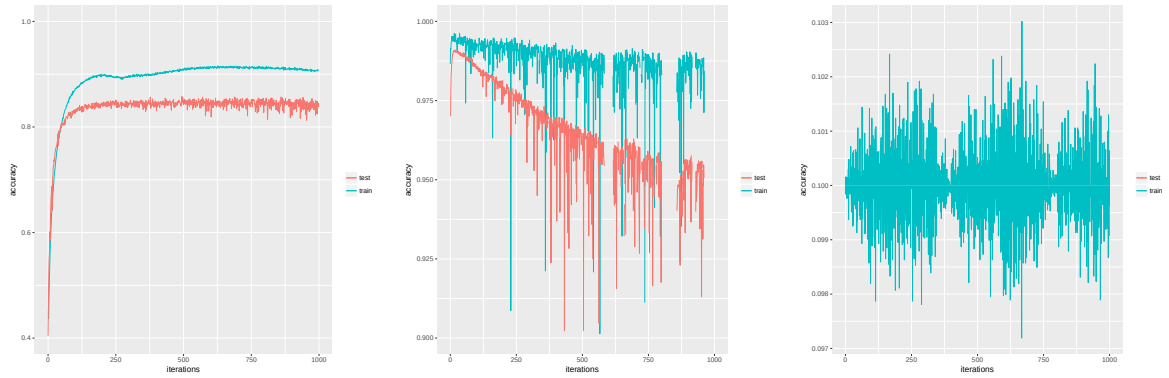
Figura 5.8: Sobreaprendizaje en MNIST

funcionar mejor o peor, aprender en mayor o menor medida, pero nunca se había producido sobreaprendizaje.

Ratio de aprendizaje Durante el estudio del impacto del ratio de aprendizaje en las redes CNN se han encontrado dos cuestiones interesantes: la diferencia de los ratios con respecto a los MLPs y el decaimiento del mismo en la función de optimización.

Se ha podido observar que el ratio de aprendizaje en las redes convolucionales debía ser mucho más bajo que en la redes tradicionales. La motivación de esto viene de que el optimizador se usa tanto para las capas densas como para las capas de características. Y, si bien en las capas densas ratios de aprendizaje altos –por ejemplo, del orden de 0.3– pueden sonar razonables, en las capas de convolucionales estos ratios hacen que la capa no sea capaz de aprender a extraer características –lo normal son ratios de 0.0001–. Si se producen saltos bruscos en las modificaciones de los pesos en cada iteración, es muy difícil que las capas convolucionales se especialicen en ninguna característica. En cada iteración es posible que la característica esté en una posición distinta de la imagen, y como la red da poco peso a

las iteraciones anteriores y mucho al aprendizaje de esta última, tiende a sobreescribir los estados anteriores.



(a) Ratio de aprendizaje de 0.0001

(b) Ratio de aprendizaje de 0.0005

(c) Ratio de aprendizaje de 0.05

Figura 5.9: Comparación de la oscilación del error en CNNs
Gráficas realizadas con CNN de 900 neuronas en las capas densas y 1000 iteraciones de entrenamiento

Como se puede ver en la Figura 5.9, los ratios de aprendizaje tienen que ser muy pequeños, como en la Figura. De otra manera, la oscilación es muy alta, como en la Figura, y la red no es capaz de aprender de ninguna forma.

El decaimiento en la función RMSprop, como en muchas otras funciones de optimización se trata de un parámetro que controla la reducción del ratio de aprendizaje con el paso del tiempo. Es un concepto parecido al *momentum*, pero con el que no debe confundirse. El *momentum* es un parámetro que reciben algunas funciones de optimización –como el descenso de gradiente estocástico– que regula el impacto en los nuevos pesos que tiene la diferencia de los pesos en los dos estados anteriores. Esta diferencia lo que hace es incluir en la función de modificación de los pesos la "velocidad" a la que cambiaron previamente. Es otra forma de tener algo de memoria. Generalmente el *momentum* se usa con la intención de hacer una función de modificación de los pesos más suave, sin tantas oscilaciones. No es raro, de todas formas, que en muchas redes de neuronas el *momentum* se suela considerar 0.

El decaimiento, por otra parte, es un parámetro que atenúa el ratio de aprendizaje con el paso del tiempo. Esto quiere decir que cuantas más iteraciones pasen, más se reduce el ratio de aprendizaje. Depende un poco de la función de decaimiento que se use el impacto real que tiene. En el caso de RMSprop, como veíamos anteriormente, el ratio de aprendizaje

se divide entre la media de los resultados del descenso de gradiente en iteraciones anteriores. El decaimiento afecta a la fuerza de esta media de los descensos de gradiente.

Debido a esto, los valores que toma el decaimiento suelen ser extremadamente bajos. Con valores más altos, la red converge antes, lo cual puede ser positivo en según que tareas, pero en este caso es –en general– bastante negativo. Convergencias rápidas debido a un ratio de aprendizaje muy bajo hace que sea mucho más fácil detener el aprendizaje en un mínimo local pensando que se ha encontrado una buena solución. Y en el reconocimiento de imágenes, por la propia naturaleza de la tarea, desconocemos la función de error.

Número de neuronas El estudio de la comparación del número de neuronas no ha sido especialmente significativo. Si que es cierto que, por la forma de las CNNs, el número de neuronas que se ha modificado ha sido el de la capa densa –que es la encargada de realizar la clasificación–. Sin embargo, gran parte del peso de las CNNs, sobre todo en tareas complejas –como el CIFAR– está en las capas convolucionales, por lo que es normal que no se hayan notado cambios notables.

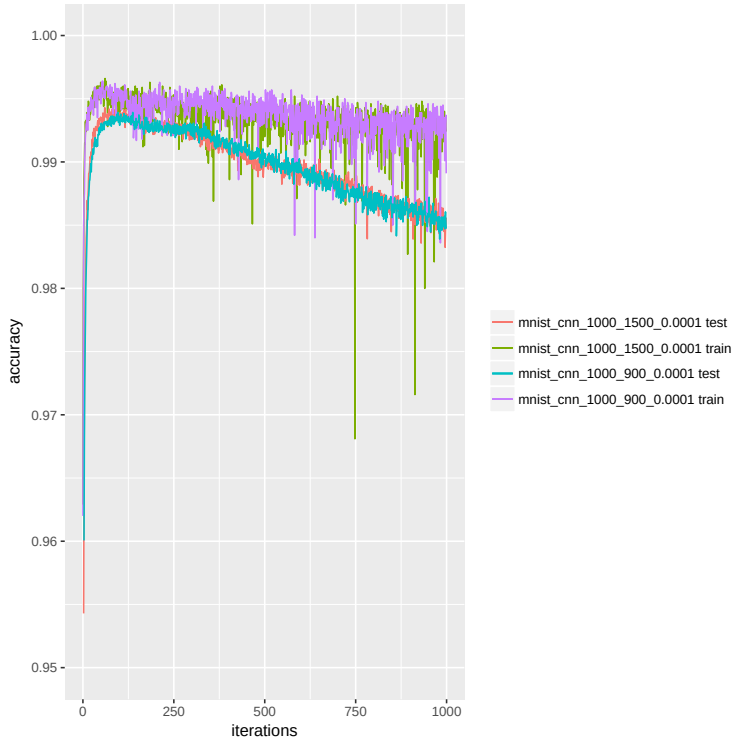


Figura 5.10: Comparación del número de neuronas en MNIST realizado sobre dos redes CNN con un aprendizaje de 0.0001, con 900 y 1500 neuronas

En la Figura 5.10 podemos ver una comparativa directa entre dos redes con diferentes número de neuronas en el conjunto de datos MNIST. Como se puede apreciar las diferencias son mínimas ya que, recordemos, en este conjunto los resultados eran muy similares. Sin embargo, si se puede ver hasta cierto punto que con 900 neuronas el aprendizaje funciona mejor. Tanto en entrenamiento como en test consigue resultados muy bueno y la oscilación del error no es tan alta como con 1500 neuronas. Podemos ver –gráfica verde– las oscilaciones de la red con 1500 neuronas, que son muy grandes constantemente.

Cabe destacar, así mismo, el sobreaprendizaje que se produce en este conjunto de datos. Como ya se comentaba en la introducción de la sección, es bastante sorprendente, dado que en este tipo de problemas no suele ser un inconveniente. Si se presta atención a la gráfica se puede ver que el descenso de la precisión en test de la red con 900 neuronas no es tan rápido. Este comportamiento lleva a pensar que más allá de las 900 neuronas la red no funciona tan bien en este conjunto de datos, y tiende a sobre aprender. No es demasiado sorprendente esta constatación, pues ya en la literatura del tema se menciona 900 neuronas como el "número mágico" para las redes sobre MNIST.

En cualquier caso, se han realizado algunos experimentos aumentando los ratios de aprendizaje para descartar que este sobraprendizaje sea en realidad la red que no es capaz de aprender. Sin embargo, al aumentar esto ratios, si bien la precisión test caía en picado –en muchas ocasiones casi hasta el 0–, también caía la precisión en entrenamiento a la par. Esto nos indica que no es el mismo caso, pues en estas gráficas se puede ver que el entrenamiento se mantiene más o menos estable, y si se reduce es levemente, no con la misma velocidad que el conjunto de test.

En el estudio sobre el conjunto CIFAR –Figura 5.11–, sin embargo, se han obtenido resultados más comprensibles y esperados. Salvando las oscilaciones y la pequeña caída que tiene la red debidas al ratio de aprendizaje, la red con más neuronas parece que consigue mejor precisión. La red con 1500 neuronas en entrenamiento claramente obtiene mejores resultados en entrenamiento que con 900 neuronas. Esto es de esperar, pues al ser un problema diferente, el número de neuronas debe ser calculado para él –sin tener en cuenta el MNIST–. Atendiendo a las evoluciones en test, sin embargo, la oscilación de la precisión en la red con 1500 neuronas es mucho más alta que con un menor número de neuronas.

Realizando diferentes pruebas, se ha podido concluir que en las redes convolucionales el ratio de aprendizaje y el número de neuronas de la capa densa está mucho más relacionado que en otras arquitecturas. Al afectar el ratio a toda la red –capas convolucionales incluidas–

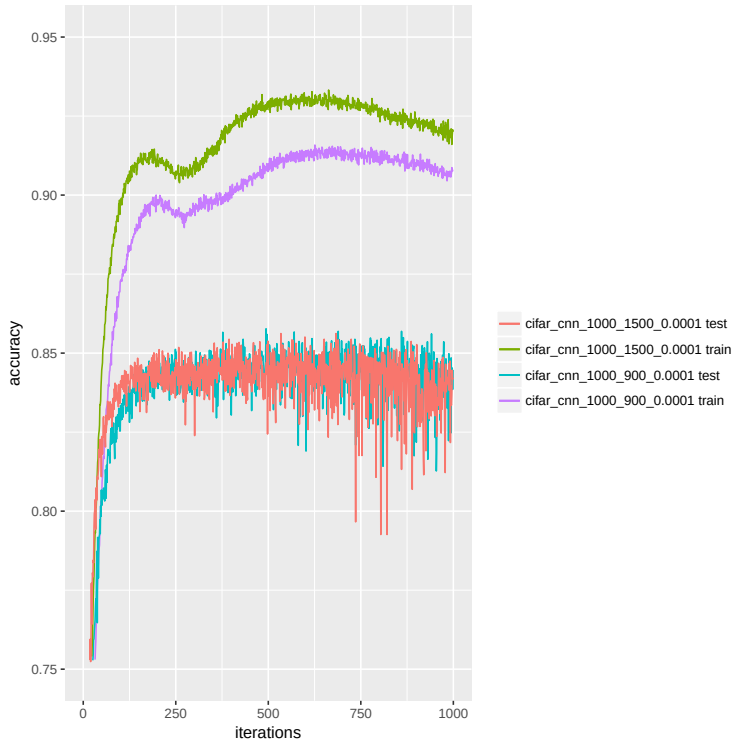


Figura 5.11: Comparación del número de neuronas en CIFAR realizado sobre dos redes CNN con un aprendizaje de 0.0001, con 900 y 1500 neuronas en cada capa densa

pero solo modificar los tamaños de la red de clasificación –la densa–, estamos provocando velocidades de aprendizaje diferentes de las esperadas en las capas convolucionales. Esto es debido a que, con el optimizador RMSprop, la cantidad de neuronas en cada capa afecta al impacto del ratio de aprendizaje en la misma.

Función de activación El estudio de la función de activación, que a priori parecía que iba a arrojar poca luz, ha constatado la problemática del sobreaprendizaje, principalmente. En ambos casos se han realizado experimentos de funciones ReLU –las que se han usado por defecto en las CNNs– contra la función sigmoide en las dos capas densas.

Atendiendo a la Figura 5.12 podemos extraer varias conclusiones. La más deseable, sin duda, es la ausencia de sobreaprendizaje al utilizar una función sigmoide. Como se puede apreciar, la oscilación del error es similar –causada por el ratio de aprendizaje– pero el conjunto de entrenamiento avanza a la par que el conjunto de test. Concretamente, en esta Figura se ve como la precisión de ambos conjuntos se superpone. Esto, hilando con lo mencionado en la sección anterior, tiene más sentido pues con la función sigmoide el error

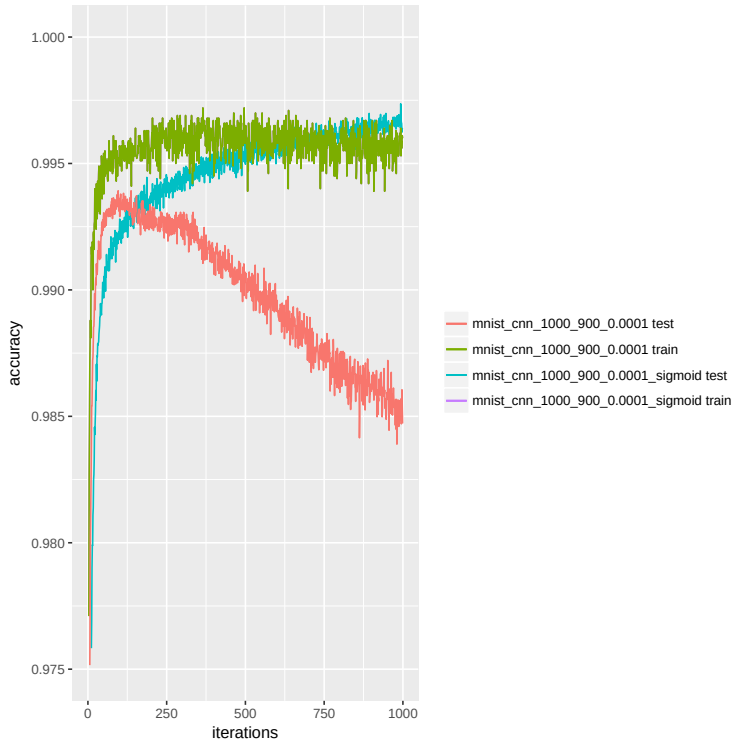


Figura 5.12: Comparación de la función de activación en MNIST entre sigmoide y ReLU, realizada sobre dos redes CNN con 900 neuronas en cada capa densa y un aprendizaje de 0.0001

se debería propagar de forma más lenta a las primeras capas de la red, haciendo que el aprendizaje sea algo menor, pero eliminando el sobreaprendizaje.

Por otra parte, también debido a la actualización de pesos en base a la sigmoide, el aprendizaje de la red es mucho más deseable. Como la propagación del error más débil evita que la red sobreaprenda, las precisiones generales son mucho mejores. De esta forma, aunque la precisión en entrenamiento con ReLU descienda frente a la de la sigmoide, la precisión en test es notablemente mejor. Esto hace que la red tenga una precisión general mejor con la función sigmoide en las capas densas que con la función ReLU.

Además –y quizás sea lo más interesante– con esta configuración se obtiene en test el mejor resultado obtenido durante toda la experimentación. Con una precisión de 0.996 es un resultado muy bueno ya que el mejor obtenido hasta el momento es de 0.9973 –mejorado hasta 0.9979 solamente por una paralelización de cinco CNNs–.

Estudiando la misma variación –el cambio de ReLU por sigmoide en capas densas– , como se puede ver en la Figura 5.13, se ve que los resultados son algo diferentes. En

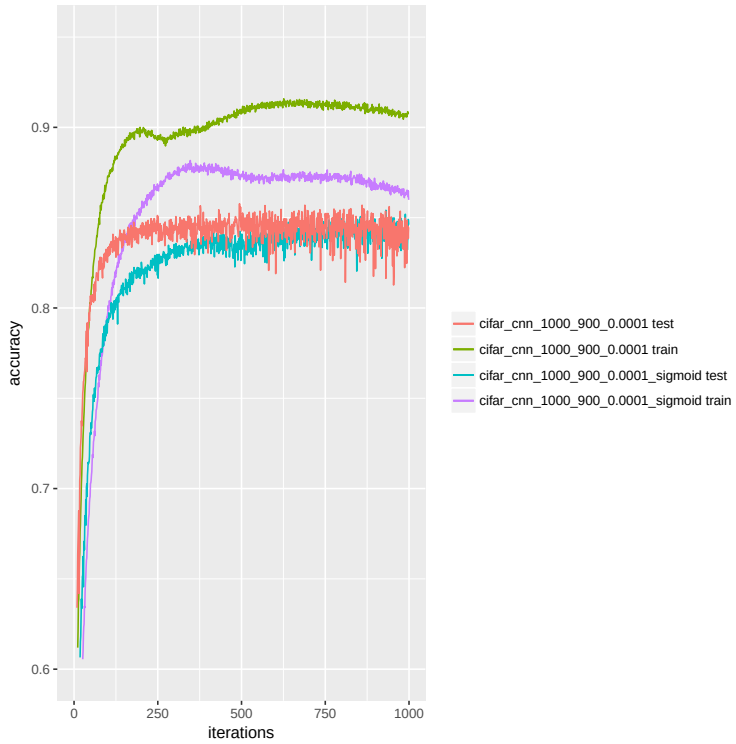


Figura 5.13: Comparación de la función de activación en CIFAR entre sigmoide y ReLU, realizada sobre dos redes CNN con 900 neuronas en cada capa densa y un aprendizaje de 0.0001

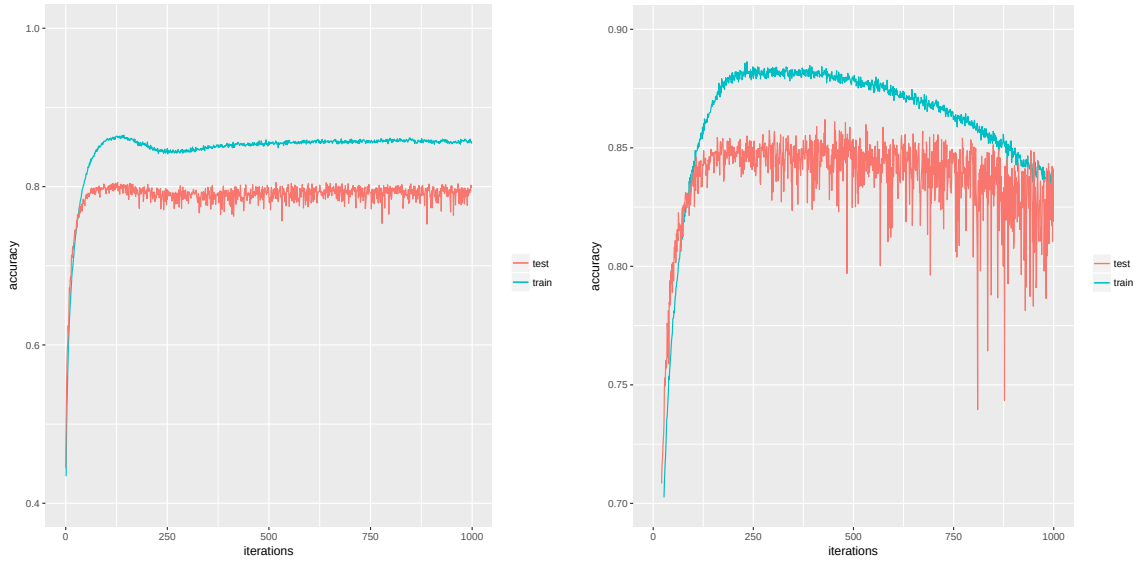
este caso no hay ningún tipo de sobreaprendizaje que corregir, pero además se ve que los resultados no son demasiado concluyentes. El aprendizaje en entrenamiento es más preciso usando funciones ReLU, aunque el aprendizaje en test se mantiene prácticamente a la par con ambas funciones.

Si cabe resaltar el comportamiento que tiene la red ReLU con ese valle en torno a la iteración 200, del que luego se recupera. Es algo que puede ser achacable a la función de error que se trata de optimizar en este conjunto de datos, aunque no es preocupante pues el conjunto de test no presenta sobreaprendizaje. Sin embargo, con la función sigmoide ese valle no existe, pero sin embargo si se ve una tendencia de bajada, como de no aprendizaje bastante sorprendente.

Número de capas convolucionales Este apartado del estudio viene motivado esencialmente por las múltiples referencias encontradas en la literatura respecto a que con más de tres capas convolucionales las CNN empezaban a funcionar mal. A priori suena algo extraño.

Si parece tener sentido que haya que encontrar un equilibrio en el número de capas, igual que en una red de neuronas tradicional no mejora el resultado por añadir más capas de forma indefinida. Sin embargo, da la sensación de que al tratarse de una técnica de *Deep Learning*, lo razonable es pensar que la arquitectura debería contar con muchas capas –pensando dos o tres como un número razonablemente pequeño de capas–.

Las conclusiones de esta sección deben tomarse con cuidado, pues los experimentos realizados aquí competen solamente al dominio del CIFAR –los experimentos de MNIST no han sido particularmente significativos–. Esto quiere decir que las conclusiones aquí extraídas, aunque tengan un carácter algo general, quedan circunscritas al dominio del CIFAR. Por ejemplo, que tres capas convolucionales –con la configuración particular utilizada– sean demasiadas capas en este caso no quiere decir que sea aplicable a todas las redes convolucionales.



(a) CNN con una capa convolucional, sin grandes cambios

(b) CNN con tres capas convolucionales donde se ve como la red no aprende

Figura 5.14: Variación del número de capas convolucionales en CIFAR, sobre una red CNN de 900 neuronas en cada capa densa y un ratio de aprendizaje de 0.0001

Como podemos ver en la Figura 5.14b, tres es un número de capas suficiente –al menos en el conjunto CIFAR– para que la red no solo deje de aprender, si no que empiece a "desaprender". Con el paso de las iteraciones, la red comienza a perder precisión en el conjunto de entrenamiento rápidamente. En el conjunto de test, el descenso es mucho más leve, aunque

la precisión empieza a oscilar gravemente. Los resultados al inicio del entrenamiento no despreciables; tiene una buena precisión. Sin embargo, el hecho de tener que controlar que la red empiece a desaprender es algo desaconsejable para un buen modelo.

En el conjunto de CIFAR se han observado reacciones similares con la misma configuración: la mejor configuración de CNN conseguida pero utilizando tres capas convolucionales en lugar de dos. El conjunto de entrenamiento también se solapa con el de test e incluso empieza a tener menos precisión que este. No se han incluido gráficas puesto que las diferencias son bastante pequeñas.

La causa se debe al propio funcionamiento de las CNN. Cuando se tienen dos capas convolucionales, la extracción de la segunda capa convolucional se realiza sobre los resultados de la primera capa convolucional. Esto es, de forma intuitiva, que extrae características de la primera extracción de características. Esto es muy interesante en ciertos problemas, pues permite que la primera capa extraiga características más generales y la segunda puntos característicos de las mismas. Si apilamos una tercera capa, se está realizando una extracción de características sobre la segunda extracción de características.

Recordemos además que las capas covolucionales se "desplazaban" por la imagen, de forma que cada neurona estudiaba un pequeño trozo de la imagen general, pudiendo solaparse. La segunda capa convolucional recorrerá esos trozos de imagen ya generados por la primera capa. Con una tercera capa, la granularidad es muy alta, prácticamente se están extrayendo características de cada pequeño trozo de la imagen. Esto es la receta de un cóctel de sobreaprendizaje. Si a esto se le suma que las imágenes van variado constantemente, el resultado es precisamente lo que se ve en la Figura. No solo es que la red sobreajuste o no aprenda bien, es que olvida los patrones previos deja de aprender.

Con una capa, sin embargo, como podemos ver en la Figura 5.14a no se ven cambios significativos respecto al comportamiento normal, comparando por ejemplo con la Figura 5.11. Las precisiones son similares, aunque con una capa cae un poco. Este comportamiento, sin embargo, es esperable pues con una única capa convolucional es de suponer que la precisión será menor. Naturalmente, una capa extrae muchas menos características que dos, por lo que el aprendizaje nos será tan bueno.

5.3. Comparativa entre MLP y CNN

En esta sección se intentará realizar un estudio sobre las diferencias en ambos conjuntos de las redes de neuronas multicapa y las redes de neuronas convolucionales. Como ya se ha menciona en alguna ocasión a lo largo del trabajo, la comparativa en el caso del conjunto de datos del MNIST es algo compleja pues las diferencias de precisión son mínimas. Por este motivo, el peso de ambos conjuntos no es totalmente igualitario, aunque sí equitativo. En esta cuestión un conjunto como el CIFAR nos ofrece mucha más información –si bien el MNIST es un buen punto de partida–.

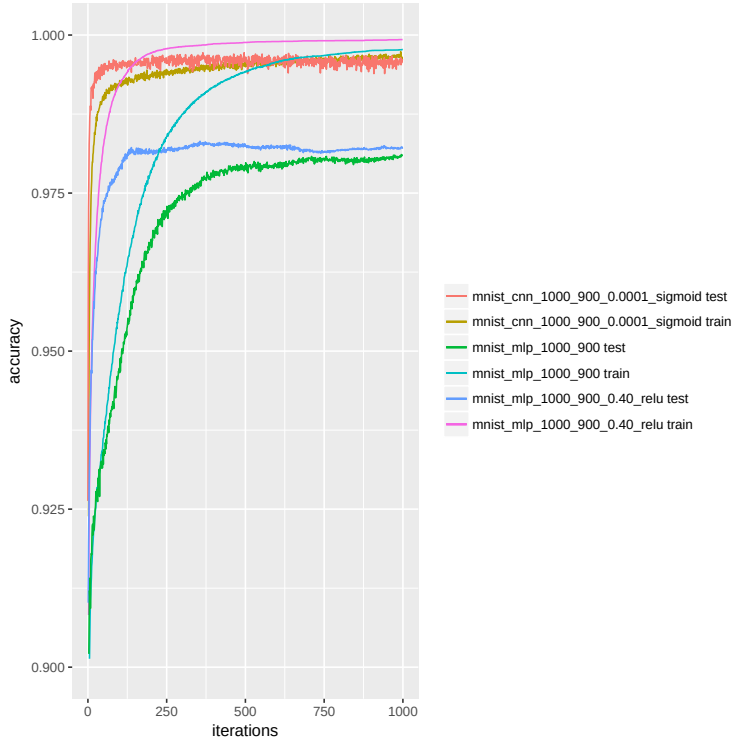


Figura 5.15: Comparación en MNIST de MLP con 900 neuronas en las capas ocultas, 0.4 como ratio de aprendizaje y función de activación sigmoide –verde– y ReLU –azul oscuro–; con una red CNN con 900 neuronas en cada capa densa y 0.0001 de ratio de aprendizaje, con función de activación sigmoide –rojo–

Usando la Figura 5.15 podemos ver que es un conjunto de datos donde una red con forma de MLP ya consigue por si misma mucha precisión. En este sentido, nos da algunas pista sobre la dificultad del problema. Y sobre que quizás el uso de CNN, si bien pueda mejorar algo el resultado, sea una potencia excesiva. La configuración de las redes

convolucionales es, por norma, más compleja que las redes multicapa. Es más difícil en el sentido de que cuenta con muchos más parámetros que ajustar y comportamientos, en ocasiones, complicados de comprender de forma intuitiva.

De los experimentos realizados en MNIST, los mejores resultados –en entrenamiento– se han obtenido con las redes multicapa. Es interesante que la CNN no haya llegado a tener un mejor resultado en entrenamiento –salvo el experimento de la Figura 5.13–. Sin embargo, en medida general de precisión, la CNN aproxima mucho mejor el problema que los MLP. Sus precisiones de test –azul oscuro y verde– quedan bastante por debajo de la precisión de test de la CNN. Ya se esperaban lograr mejores resultados con la CNN, aunque es notable el hecho de que en entrenamiento se comporten mejor los MLP.

De hecho, en la literatura del tema –estudios sobre MNIST–, los mejores resultados parten de agrupaciones de redes convolucionales, o de capas convolucionales apiladas sobre otro tipo de algoritmos –como SVM o KNN–. Las diferencias, naturalmente, son de milésimas de precisión, pero se debe a que el problema con una red multicapa ya tiene unos resultados muy aceptables. Aún teniendo en cuenta que el margen en MNIST es demasiado pequeño para sacar conclusiones generales, si parece que el problema puede ser abordado por ambos tipos de redes, entrando en valor otras características: eficiencia, recursos...

En cuanto al sobreaprendizaje que se veía previamente, en el MNIST es razonable encontrarlo en CNNs. Las CNNs son redes pensadas para realizar un preprocesado en la propia red que extraiga características comunes. Normalmente cada célula de la capa convolucional se centra más en la característica concreta que en la localización de la misma. Sin embargo, por la forma que tiene el MNIST –imágenes de números siempre en las mismas posiciones–, es un conjunto mucho más dado a que una red que solamente reconoce características sobreaprenda. En el conjunto CIFAR, por el contrario, como las imágenes representan objetos más generales –coches, perros– que no siempre están en la misma posición, funcionan mejor estas redes y es más difícil que se den sobreaprendizajes.

Estudiando el conjunto CIFAR, los resultados se acercan a lo esperado. Como se ve en la Figura 5.16, la red que mejor se adapta a los patrones del CIFAR es la CNN con ReLU. La CNN sigmoide, como ya se mencionó anteriormente ha tenido unos resultados algo peores, por lo que se ha escogido la ReLU. Los MLPs no alcanzan la precisión de la CNN ni de lejos.

El MLP con función de activación sigmoide tiene los peores resultados con diferencia. Esto tiene sentido, y encaja con todo lo estudiado previamente. CIFAR es un conjunto

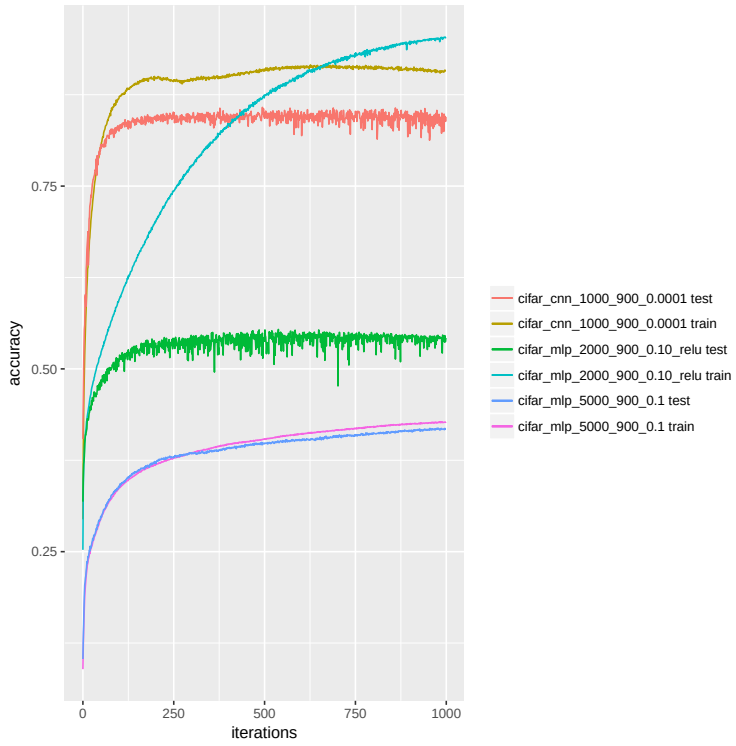


Figura 5.16: Comparación en CIFAR de MLP con 900 neuronas en las capas ocultas, 0.1 como ratio de aprendizaje y función de activación sigmoide –azul oscuro– y ReLU –verde–; con una red CNN con 900 neuronas en cada capa densa y 0.0001 de ratio de aprendizaje, con función de activación ReLU –rojo–

de datos más complejo que MNIST y es normal que un MLP no sepa interpretarlo. En la literatura hay múltiples referencias a estructuras alternativas para problemas de reconocimiento de imágenes complejos si se desea usar MLP. Estas son arquitecturas basadas en realizar un preprocesado previo, bien mediante técnicas tradicionales de extracción de características, o bien mediante las propias

Por otro lado, la red multicapa utilizando funciones de activación ReLU obtiene mejores resultados que la anterior. Como ya mencionamos previamente, en problemas complejos ReLU suele funcionar mejor pues propaga el error con más fidelidad, aunque depende mucho del tipo de función que se desee aproximar. En todo caso, queda lejos de los resultados de la CNN, atendiendo naturalmente a su resultado en test. Recordemos que, aunque la precisión en test sea sorprendentemente buena, debemos atender a su precisión en test para estar seguros de su eficiencia.

De esta forma, los resultados en CIFAR son mucho mejores con una red CNN,

llegando a los 0.85 de precisión en test. Las capas de extracción de características propias de las CNN se han demostrado muy útiles, potenciando la efectividad de estas arquitecturas en problemas complejos de reconocimiento de imágenes.

Decididos a fijar un umbral a partir del cual utilizar CNNs, parece que es precisamente la complejidad y forma del problema lo que puede darnos una pista. Naturalmente, para confirmar o desmentir cualquier hipótesis en este ámbito deberá realizarse la correspondiente dosis de experimentación. Sin embargo, dado que CIFAR es un conjunto de imágenes a color y MNIST no, parece que esa es una de las principales diferencias. CIFAR codifica esos colores de forma que se tiene una matriz tridimensional, con las dos primeras dimensiones para indicar el alto y el ancho, y la tercera para indicar el canal de color. Este tipo de estructuras –no exactamente así, pero de ese tipo– encajan muy bien con las CNN, puesto que reciben como entrada directamente ese tipo de matriz, de la que extraen las características. Como yuxtaposición, las redes con arquitectura MLP, reciben un único vector secuencial.

De esta forma, cualquier transformación de la matriz tridimensional a un vector unidimensional va a hacernos perder información. Bien en el sentido de tener las mismas posiciones desplazadas, o bien por tener –si se transforma así– los tres canales consecutivos. Sin embargo, con entradas como el MNIST, al tener un único canal de color –escala de grises– da igual que la entrada sea unidimensional, pues no se tendrán los tres colores.

6. Conclusiones y trabajo futuro

6.1. Conclusiones

Tras un análisis detallado de los experimentos realizados conviene realizar una reflexión sobre los objetivos marcados inicialmente para el desarrollo de este trabajo y su consecución. Así mismo, esta sección sirve también para agrupar las principales conclusiones extraídas del estudio, de forma que puedan ser repasadas de forma independiente, recurriendo a la fuente solamente para contrastar datos.

6.1.1. Características más importantes y su impacto en las redes

A modo de resumen y recapitulación, los parámetros que más impacto han tenido se exponen a continuación. Las referencias completas se podrán, naturalmente, repasar en la Sección 5.1.1. De todos los parámetros estudiados, los más impactantes han sido los siguientes.

Ratio de aprendizaje El ratio de aprendizaje se ha demostrado un parámetro, desapercibido inicialmente, con un impacto brutal en las redes. Como se veía en la Figura 5.9, hasta que punto el ratio de aprendizaje influye que puede hacer que una red no aprenda absolutamente nada, a pesar de tener muy buena configuración en el resto de parámetros.

Además, este parámetro está ligado en gran medida a la función de optimización elegida en el aprendizaje, así como a la forma y cantidad de neuronas de las capas. Esto lo hace un parámetro sencillo de variar, pero con un peso bastante importante en la configuración de la red. También hay que tener cuidado con ella, pues al estar tan ligada a otros parámetros nos puede dar falsos negativos, provocando un mal comportamiento en la red al variar estos parámetros de forma independiente.

La ventaja es que los efectos de un ratio de aprendizaje alto son relativamente fáciles de detectar, pues normalmente conllevan una oscilación del error muy grande. En otras ocasiones, sin embargo, con una mala configuración del ratio no se ve tanta oscilación –debido a otros parámetros– pero causa una convergencia prematura no deseada.

Funciones de activación El estudio de las funciones de activación ha resultado bastante interesante, aunque las conclusiones han sido algo tibias. Se ha podido ver lo importante que era este parámetro, pues tiene un gran impacto en el aprendizaje de la red. Los tiempos de convergencia, las precisiones en entrenamiento y test, así como el aprendizaje general de la red está muy ligado a las funciones de activación que se escojan.

Sin embargo, las conclusiones han sido algo tibias pues no se ha ni acercado a una norma general. Lo más que se ha podido deducir es que es un parámetro muy ligado a la función de optimización y de error que se utilicen, así como al tipo de problema. Por ejemplo, ya se comentaba anteriormente que es muy usual aplicar una función de activación softmax en la última capa –la capa de salida– en tareas de clasificación. Esto ayuda notablemente a que la red "entienda" mejor los resultados y el aprendizaje sea mayor. Así mismo, también hemos visto que con redes muy grades –muchas capas– o con redes convolucionales es usual, y mucho más práctico, la utilización de funciones ReLU en lugar de sigmoidales. Esto se debe a que las ReLU tienen mucha menos atenuación del error con el paso de las capas en el backpropagation.

A parte de estas ideas de carácter general, no se ha podido detectar un patrón más claro de qué situaciones son susceptibles –o no– de utilizar funciones de activación sigmoidales, tangenciales o ReLU. De modo que para encontrar la mejor función de activación debe recurrirse a la experimentación con el problema concreto que se quiera abordar. No obstante, aunque no se haya logrado una regla mágica, queda recogida esta colección de conclusiones para tener en cuenta de cara a la experimentación.

Arquitectura de la red Mención especial requiere este punto, pues no puede considerarse un parámetro en sí mismo, aunque si es especialmente influyente para la adecuación de la red al problema –como parece obvio, por otra parte–. Tampoco pretendo detenerme más tiempo en este apartado, pues está íntimamente relacionado con la Sección 6.1.3, donde se tratará este tema también.

La arquitectura de la red, no entendiéndola solamente por si es una red convolucional o una red recurrente, si no por la propia estructura de la red es un parámetro muy importante a tener en cuenta. Por ejemplo, una vez decidido usar una red convolucional, no tiene el mismo impacto ni comportamiento que la red cuente con una capa convolucional que el hecho de que tenga tres, o que la cantidad de neuronas en las capas densas sean o no las mismas.

Estos parámetros afectan mucho al aprendizaje de la red y están terriblemente liga-

dos al tipo de problema, dejando solamente la oportunidad de mejorar este aspecto mediante la experimentación con el conjunto de datos. Sin embargo, si se ha llegado a algunas conclusiones interesantes. Tras algunos experimentos variando la cantidad de capas convolucionales se aprecia que más no siempre significa mejor. Concretamente, a partir de tres capas convolucionales, normalmente las precisiones obtenidas caen en picado. Esto se deduce de la dificultad de propagar el error hacia atrás en este tipo de redes –incluso con funciones ReLU y optimizadores especiales, como RMSprop. Igual de interesante es el hecho de que más de dos capas totalmente conectadas o una cantidad de neuronas que no sea igual en ambas capas empieza a producir una caída en el rendimiento de la red –de forma general– bastante notable.

Como ya hemos visto en el análisis de CNN, el impacto que puede tener la arquitectura de la red es muy importante. Uno de los principales motivos por los que las CNN funcionan tan bien con los problemas de reconocimiento de imágenes es precisamente por el reconocimiento de características de las capas convolucionales. Si esa extracción de características está mal integrada en la red, puede ser desastroso para la misma. Incluso con una configuración de parámetros correcta, una capa extra donde no debe puede tener mucho impacto en la red.

6.1.2. Comparación del rendimiento de MLP vs CNN

Las conclusiones en esta comparación han ido en la línea de lo esperado, en base a la literatura existente. Por una parte, las redes CNN han funcionado mejor, en general, que los MLP, aunque en el conjunto de MNIST ha estado muy reñido. Esto tiene mucho sentido dada la capa de preprocesado previo que tienen incluidas las CNN con las que no cuentan los MLP. Incluso, en CIFAR la precisión de las CNN frente a los MLP ha sido brutal.

Teniendo en cuenta esto, en problemas como el del MNIST hay que valorar dos objetivos: por una parte el objetivo investigador, según el cual se pretende lograr la mejor precisión, en cuyo caso las CNN son la mejor elección. Y por otra parte, cabe contemplar que si el objetivo fuese más productivo –tener un sistema en un entorno real– los MLP probablemente fueran más interesantes, pues aunque tienen algo menos de precisión, tardan mucho menos en ejecutarse y requieren menos recursos. Al final, otro factor que entra en juego, más allá de cuál arquitectura funciona mejor o peor objetivamente, es una cuestión de diseño: ¿con qué fin estoy diseñando este modelo? ¿Pretendo que sea el mejor? ¿O pretendo que sea el modelo más eficiente? Y las medidas de precisión y eficiencia vienen, naturalmente,

determinadas por el problema que se plantea.

Por otro lado, a un nivel más concreto, se puede definir una regla para utilizar o no las CNN: las CNN se aplican en los tipos de datos en los que se pueda representar la información como una matriz bidimensional o tridimensional, mientras que los MLP solo si se puede presentar como unidimensional. Esto se debe a que las CNN para funcionar reciben como entrada una matriz, generalmente bi o tridimensional. A partir de esa matriz pueden aplicar las capas convolucionales, pero sin tener los datos expuestos de esta forma, su funcionamiento se resiente.

Sin embargo, los MLP reciben un vector unidimensional como entrada. Esto quiere decir que los datos tienen que poder transformarse a eso de una forma coherente. Una solución típica suele ser, sencillamente, concatenar los datos de la matriz bidimensional. Está practica lo que suele causar es romper las relaciones espaciales del problema, dificultando la labor a la red neuronal. Cuanto más complejo sea el problema y más relaciones espaciales tenga, más impacto se notará en la precisión de la red.

6.1.3. Adecuación general de la arquitectura al conjunto de datos

Después de todo lo estudiado a lo largo de este trabajo, parece obvio que la adecuación de la arquitectura al conjunto de datos depende de múltiples factores, generalmente ajenos a la red y más propios del conjunto que a otra cosa. Sin embargo, se pueden definir ciertas normas o ideas generales que pueden guiar nuestras decisiones de forma más inteligente. Nótese que todas estas indicaciones son de carácter general y de ninguna forma son siempre válidas ni sientan ningún tipo cátedra, todas obtenidas mediante la experiencia y seguramente mejorables en muchos aspectos.

Problemas pequeños o con pocas variables de entrada, así como pocas variables de salida, pueden ser resueltos con relativa facilidad por redes de neuronas tradicionales –MLPs o técnicas no supervisadas– de forma mucho más rápida que utilizando Deep Learning. De hecho, en problemas pequeños utilizar Deep Learning es contraproducente pues usualmente don mucho más costosas –computacionalmente hablando– estas técnicas respecto a redes tradicionales, muchas veces con una sola capa y pocas neuronas.

Así mismo, problemas que tengan una cantidad de datos pequeños, es decir, con conjuntos de datos sin demasiados datos, no son demasiado susceptibles de ser resueltos con facilidad con Deep Learning tampoco. Con una cantidad de datos pequeña la red –convolucional o usando la técnica que sea– probablemente no sea capaz de aprender lo suficiente de esos

datos. Si se tienen muchos recursos, puede ser interesante aún así intentarlo, algunas veces los resultados aplicando Deep Learning a conjuntos pequeños son sorprendentes.

Para explotar la máxima potencia de estas técnicas, sin embargo, se necesita una cantidad muy grande de datos –casi masiva– y que presenten retos inabarcables mediante otras técnicas. Bien sea por patrones complejos o por cantidad de variables de entrada, las técnicas de Deep Learning se comportan muy bien con conjuntos de datos muy grandes. De hecho, parte del boom que ha sufrido el Deep Learning actualmente se basa en que muchas empresas y grandes compañías –como Google o Facebook, por nombrar algunas, aunque hay muchas más– están haciendo uso de la ingente cantidad de datos que llevan almacenando durante años para entrenar estos sistemas, lo que hace que funcionen extremadamente bien.

Por ejemplo, que mejor entrenamiento pueden sufrir los algoritmos de etiquetado automático de Facebook que su propia base de datos de imágenes. Una red social que construye su atractivo precisamente en la publicación de fotos y el etiquetado de personas y lugares en ellas presenta una forma muy orgánica –pero potente– de generar un dataset inmenso. O entrenar algoritmos de etiquetado automático de vídeos con la base de datos de vídeos más grande del mundo –YouTube–.

6.2. Trabajo futuro

Tras todo lo expuesto en este trabajo puede verse que, si bien se han resuelto las cuestiones planteadas en gran medida, sigue quedando mucho campo de mejora en este ámbito. El tema de las redes de neuronas en general, y del Deep Learning en particular presenta grandes desafíos actualmente para la comunidad científica e investigadora.

Si bien el acceso antes estaba muy restringido y la experimentación era muy compleja, en el estado actual la cantidad de herramientas y recursos disponibles de forma gratuita y libre son casi infinitos. Esto, sumado a que el acceso a equipos relativamente ponentes es sencillo facilita mucho la investigación aún sin pertenecer a ningún gran centro investigador.

Centrado en el trabajo aquí realizado, aún hay muchas incógnitas por responder. Por ejemplo, uno de los experimentos futuros que han quedado pendientes es realizar toda esta experimentación con otros tipos de conjuntos –a priori, de imagen como el SVHN, pero incluso podría plantearse estudiar otros ámbitos–. Realizar esta experimentación –no clonada, si no adaptada– en conjuntos de datos de otras disciplinas, como el procesado del lenguaje natural o incluso predicción de series temporales –una tarea en la que las redes de neuronas

tradicionales son razonablemente buenas– podría ser extremadamente enriquecedor. También ofrecería nuevas perspectivas y arrojaría nuevos resultados.

Por otro lado, aún con los conjuntos de datos actuales, también sería muy interesante probar la efectividad de otras técnicas actuales, como por ejemplo preprocesados con autoencoders en lugar de capas convolucionales, utilizar RNNs o incluso probar la efectividad de las GAN [26] –un tipo de redes de neuronas utilizadas en aprendizaje no supervisado con el objetivo de generar contenido similar al estudiado–.

Los retos y posibilidades son muchos y muy excitantes. Se puede considerar que el Deep Learning está naciendo, por lo que aún nos queda por ver su gran auge y la expansión de su uso.

7. Planificación y presupuesto

7.1. Planificación

Para facilitar el seguimiento de los diagramas mostrados en esta sección, y poder detallar la información expuesta, se adjunta una tabla con la descripción de los procesos de forma extendica, y un diagrama Gantt para facilitar el seguimiento de los mismos.

Las semanas implican un trabajo diario (de lunes a viernes) de una media de 3 horas al día. Las tareas 6 y 7 se solapan parcialmente ya que dado que puede compaginarse el estudio y realización de la experimentación con la elaboración de la memoria.

Cuadro 7.1: Planificación

Id	Tarea	Duración	Fecha inicio	Fecha fin
1	Definición del problema	2 semanas	13/02/2017	25/02/2017
2	Elección de las herramientas de desarrollo	2 semanas	27/02/2017	11/03/2017
3	Investigación sobre las técnicas y datasets	3 semanas	13/03/2017	01/04/2017
4	Elaboración de la memoria	1 semanas	27/03/2017	01/04/2017
5	Implementación de los experimentos	2 semanas	03/04/2017	15/04/2017
6	Experimentación	6 semanas	17/04/2017	27/05/2017
7	Elaboración de la memoria	5 semanas	01/05/2017	03/06/2017
8	Revisión de la memoria	2 semanas	05/06/2017	17/06/2017
9	Elaboración de la presentación	1 semanas	19/06/2017	23/06/2017

Como se puede ver en la Figura 7.1, solamente son concurrentes en el tiempo las tareas de experimentación y elaboración de la memoria, ya que son dos tareas que pueden realizarse parcialmente en paralelo. Además, la tarea 4, que corresponde a una elaboración temprana de la memoria (partes como la planificación y presupuestos, herramientas utilizadas, introducción, etc), es independiente al flujo del desarrollo de los scripts de experimentación, por lo que puede avanzarse en la zona central del proyecto.

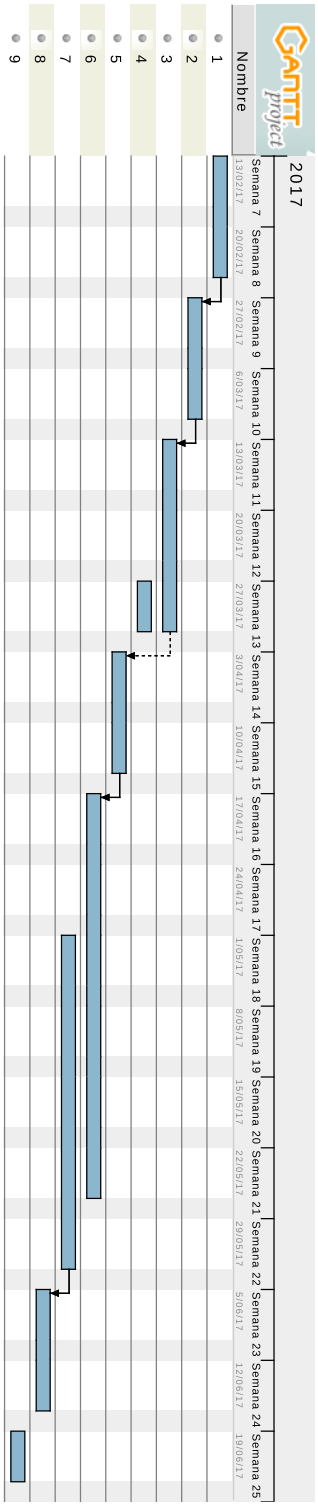


Figura 7.1: Diagrama Gantt de la planificación

En total se han utilizado 120 días, que se traducen en unas 360 horas.

7.1.1. Planificación final

Debido a algunos periodos de baja actividad finalmente fue necesario replanificar, quedando finalmente la siguiente distribución de tiempos:

Cuadro 7.2: Planificación Final

Id	Tarea	Duración	Fecha inicio	Fecha fin
1	Definición del problema	2 semanas	13/02/2017	25/02/2017
2	Elección de las herramientas de desarrollo	2 semanas	27/02/2017	11/03/2017
3	Investigación sobre las técnicas y datasets	3 semanas	13/03/2017	01/04/2017
4	Elaboración de la memoria	1 semanas	27/03/2017	01/04/2017
5	Implementación de los experimentos	2 semanas	03/04/2017	15/04/2017
6	Implementación de los experimentos	3 semanas	29/05/2017	16/06/2017
7	Experimentación	6 semanas	19/06/2017	15/09/2017
8	Repetición de parte de la experimentación	3 semanas	21/08/2017	08/09/2017
9	Elaboración de la memoria	5 semanas	31/07/2017	01/09/2017
10	Revisión de la memoria	3 semanas	04/09/2017	22/09/2017
11	Elaboración de la presentación	1 semanas	25/09/2017	29/09/2017

Los principales motivos que justificaron los retrasos y la baja actividad fue el periodo de exámenes, en el que se había estimado mucho más trabajo del que finalmente se llevó a cabo, así como el desarrollo del código de los experimentos. Este desarrollo presentó mayores dificultades de las esperadas, sobre todo al adaptar los conjuntos de datos para la realización de algunos experimentos. Además, una parte de la experimentación tuvo que repetirse, pues los resultados no habían sido válidos. El equipo de experimentación se apagó a principios de agosto, y al estar emplazado en la universidad no se pudo encender de nuevo hasta finales de agosto. Así mismo se puede ver en el Gantt (Figura 7.2) la variación de la planificación.

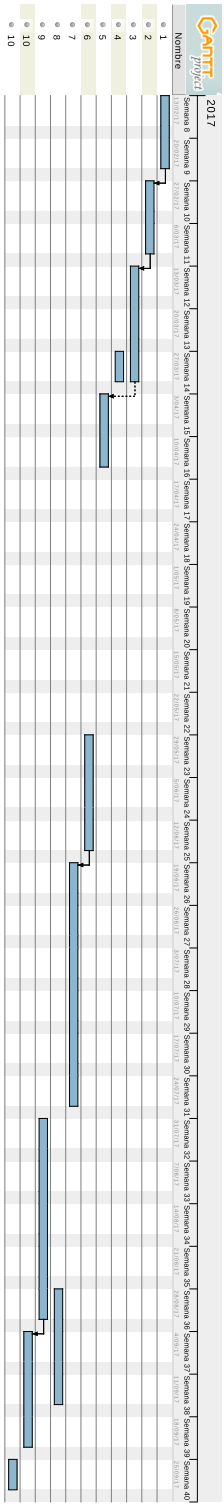


Figura 7.2: Diagrama Gantt de la planificación final

Se han utilizado finalmente 155 días (35 más de los planificados), que se traducen en unas 465 horas, unas 105 horas más de lo definido inicialmente.

7.2. Presupuesto

En el cálculo del presupuesto, al ser un proyecto experimental en el que las herramientas utilizadas (tanto *frameworks* como lenguajes de programación y herramientas) tienen una licencia libre, el coste a contemplar se basa en infraestructura (equipos) y recursos humanos exclusivamente.

7.2.1. Recursos humanos

Se ha contemplado el sueldo del investigador, pero también del supervisor del proyecto. Para este último se han calculado un 10 % de las horas totales, en tareas de revisión

- Investigador: Ocupación total de 420 horas
- Supervisor: Ocupación parcial, del 10 %, por lo que son 42 horas

Cuadro 7.3: Presupuesto: Recursos humanos

Personal	**Horas**	**Coste** (€/*hora*)	**Coste total** (€)
Investigador	465	15	6975
Supervisor	45	20	900
		Total	7875 €

7.2.2. Hardware

Se han utilizado dos equipos, uno para el desarrollo general del proyecto (el portátil) y otro para realizar la experimentación, pudiendo tenerlo encendido de forma continuada.

- Portátil: i7-6700, NVIDIA® GeForce GTX 960M 2GB, 16GB RAM
- Torre: Intel Core 2 Duo, NVIDIA® GeForce GTX 1050 4GB, 4GB RAM

Cuadro 7.4: Presupuesto: Hardware

Recurso	**Coste**	**Amortización** (%/año)	**Uso** (días/año)	**Coste total** (€)
Portatil	1099	25 %	140/360	106,85
Torre	1000	25 %	140/360	97,22
			Total	204,07 €

7.2.3. Software

El coste total del software ha sido de 0 euros ya que se ha utilizado exclusivamente software libre u *Open Source*.

- Sistema operativo: Debian GNU/Linux y Ubuntu GNU/Linux
- Lenguajes de programación: Python y R
- Frameworks: TensorFlow y Keras
- Editor de texto: LaTeX+ TexStudio
- Entorno de desarrollo: PyCharm Community Edition y VIM

7.2.4. Resumen

En base a esto podemos calcular el presupuesto total del proyecto como:

Cuadro 7.5: Presupuesto total

Concepto	**Coste** (€)
Recursos Humanos	7875
Hardware	204,07
Software	0
Total	8079,07 €

7. Referencias

[1] P. High, "Gartner: Top 10 strategic technology trends for 2014," https://www.forbes.com/sites/peterhigh/2013/10/14/gartner-top-10-strategic-technology-trends-for-2014, (Accedido el 05/06/2017).

[2] "Tensorflow website," https://www.tensorflow.org, (Accedido el 23/02/2017).

[3] "Keras website," https://keras.io/, (Accedido el 24/02/2017).

[4] Y. Lecun, C. Cortes, and C. Burges, "Mnist handwritten digit database," http://yann.lecun.com/exdb/mnist/, (Accedido el 22/02/2017).

[5] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009, ch. 3.

[6] J. C. Hay and F. Rosenblatt, *Mark I Perceptron Operators' Manual*, disponible en http://www.dtic.mil/dtic/tr/fulltext/u2/236965.pdf.

[7] G. H. y. R. W. David Rumelhart, "Learning representations by back-propagating errors," 1968, disponible en http://www.iro.umontreal.ca/~vincentp/ift3395/lectures/backprop_old.pdf.

[8] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359 – 366, 1989, disponible en http://www.sciencedirect.com/science/article/pii/0893608089900208. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0893608089900208

[9] "History and license for python," https://docs.python.org/3/license.html, (Accedido el 10/08/2017).

[10] "History and license for python," https://docs.python.org/3/license.html, (Accedido el 10/08/2017).

[11] "Tensorflow license," https://github.com/tensorflow/tensorflow/blob/master/LICENSE, (Accedido el 10/08/2017).

[12] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y, "Reading digits in natural images with unsupervised feature learning," http://ufldl.stanford.edu/housenumbers, (Accedido el 23/02/2017).

[13] "Imagenet website," http://www.image-net.org/, (Accedido el 28/02/2017).

[14] S. Shi, Q. Wang, P. Xu, and X. Ch, "Benchmarking state-of-the-art deep learning software tools," disponible en https://arxiv.org/pdf/1608.07249.pdf.

[15] "C++ website," https://isocpp.org/, (Accedido el 15/02/2017).

[16] "Python 3 programs versus c++ g++," https://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=python3&lang2=gpp, (Accedido el 23/08/2017).

[17] "Python website," https://www.tensorflow.org, (Accedido el 15/02/2017).

[18] "R website," https://www.r-project.org/about.html, (Accedido el 15/02/2017).

[19] "Theano website," http://www.deeplearning.net/software/theano/, (Accedido el 23/02/2017).

[20] "H2o website," https://www.h2o.ai/h2o/, (Accedido el 24/02/2017).

[21] W. Jaśkowski, "Theano vs. tensorflow benchmark," https://github.com/wjaskowski/tensorflow-vs-theano-benchmark/blob/master/results_cpu.csv, (Accedido el 28/08/2017).

[22] A. Izvorski, "Simple deep learning benchmark (vgg16)," https://github.com/aizvorski/vgg-benchmarks/blob/master/results/gtx_1080/INFO.md, (Accedido el 28/08/2017).

[23] "Cntk," https://github.com/Microsoft/cntk, (Accedido el 23/02/2017).

[24] "Validación de la instalación de tensorflow," https://www.tensorflow.org/install/install_linux#ValidateYourInstallation, (Accedido el 23/02/2017).

[25] B. C. Csáji, "Approximation with artificial neural networks," 01 2001.

[26] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, and D. W.-F. et al., "Generative adversarial nets," 2014, disponible en https://arxiv.org/pdf/1406.2661.pdf. [Online]. Available: https://arxiv.org/pdf/1406.2661.pdf

A. Código relativo a los experimentos

A continuación se muestra el codigo utilizado para la experimentación: cuatro clases de Python. Se ha dividido el apéndice en dos bloques –MLP y CNN– en función de la arquitectura, pues las similitudes son notables. Dentro de cada una de esas secciones se separa la clase correspondiente a cada conjunto de datos –MNIST y CIFAR–. Todas las clases tienen la misma estructura general: la función `__init__()`, que es el constructor de la función en Python y sirve para inicializar la clase y asignar los valores iniciales de la red. A continuación tenemos una función `create_model()`, encargada de construir el modelo de forma general. Este modelo será el que luego Keras –por medio de TensorFlow– "compilará". Por último existe una función `train()` que es la que se ejecutará cuando se desee entrenar ese modelo generado en la función anterior, y que contiene el verdadero entrenamiento de la red –trazas incluidas–. Por tanto, el proceso será instanciar la clase, que ejecutará la función `create_model()` y después ejecutar la función `train()`. Un ejemplo real sería:

```
# Se instancia un MLP con 900 neuronas oculatas y un ratio de
    aprendizaje de 0.4, entrenado durante 1000 iteraciones
mlp = MLP(n=1000, hidden_layer=900, learning_rate=0.4)
# Se entrena el modelo, en result se guarda un string en formato
    csv con el resultado del entrenamiento
result = mlp.train()
```

A.1. MLP

Los MLPs, al recibir como entrada un vector unidimiensional tienen que hacer un aplanamiento –con la clase `Reshape()`– en caso de que se hayan aplicado deformaciones. Para aplicar las deformaciones el input de MNIST –que es un vector unidimensional– se convierte a dos dimensiones. Por tanto, a la entrada de la red debe reconvertirse a un vector. A parte de eso el código no tiene muchas más peculiaridades, pues solamente crea la única capa densa, y luego la capa de salida.

A.1.1. MNIST

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Reshape
from keras.optimizers import SGD
from keras.utils import to_categorical
from keras.datasets import mnist

class Mlp:
    num_classes = 10
    batch_size = 128
    reshape = (784,)
    shape = (28, 28, 1,)

    def __init__(self, n, hidden_layer, learning_rate,
            image_deformation):
        self.n_iterations = n
        self.hidden_neurons = hidden_layer
        self.learning_rate = learning_rate
        self.image_deformation = image_deformation
        self.create_model()

    def create_model(self):
        self.model = Sequential()
        if self.image_deformation:
            self.model.add(Reshape(self.reshape, input_shape=self
               .shape))
        self.model.add(Dense(self.hidden_neurons, activation='
           sigmoid',
            input_shape=self.reshape))
        self.model.add(Dense(self.num_classes, activation='
           softmax'))

        self.model.compile(loss='mean_squared_error',
```

```
                        optimizer=SGD(lr=self.learning_rate),
                        metrics=['accuracy'])

    def train(self):
        (x_train, y_train), (x_test, y_test) = mnist.load_data()

        if self.image_deformation:
            x_train = x_train.reshape(60000, 28, 28, 1)
            x_test = x_test.reshape(10000, 28, 28, 1)
        else:
            x_train = x_train.reshape(60000, 784)
            x_test = x_test.reshape(10000, 784)
        x_train = x_train.astype('float32')
        x_test = x_test.astype('float32')
        x_train /= 255
        x_test /= 255

        y_train = to_categorical(y_train, self.num_classes)
        y_test = to_categorical(y_test, self.num_classes)

        if self.image_deformation:
            datagen = ImageDataGenerator(
                width_shift_range=0.1,
                height_shift_range=0.1,
            )
            datagen.fit(x_train)

            history = self.model.fit_generator(datagen.flow(
               x_train, y_train,
                    batch_size=self.batch_size),
               steps_per_epoch=x_train.shape[0] // self.
                  batch_size,
               verbose=1,
```

```
            epochs=self.n_iterations,
            validation_data=(x_test, y_test))
        else:
            history = self.model.fit(x_train, y_train,
                batch_size=self.batch_size,
                epochs=self.n_iterations,
                verbose=1,
                validation_data=(x_test, y_test))

        result = [(history.history.get('acc')[i], history.history
           .get('loss')[i],
                   history.history.get('val_acc')[i], history.
                      history.get('val_loss')[i])
                       for i in range(len(history.history.get('
                          acc')))]
        result_string = 'position,train_accuracy,train_loss,
           validation_accuracy,validation_loss'
        for position in range(len(result)):
            result_string += '\n{:d},{:1.5f},{:1.5f},{:1.5f
               },{:1.5f}'.format(
                position, result[position][0], result[position
                   ][1], result[position][2], result[position
                   ][3])

        return result_string
```

A.1.2. CIFAR

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Reshape
from keras.optimizers import SGD
from keras.utils import to_categorical
from keras.datasets import cifar10
```

```
class Mlp:
    num_classes = 10
    batch_size = 128
    reshape = (3072,)
    shape = (32, 32, 3,)

    def __init__(self, n, hidden_layer, learning_rate,
       image_deformation):
        self.n_iterations = n
        self.hidden_neurons = hidden_layer
        self.learning_rate = learning_rate
        self.image_deformation = image_deformation
        self.create_model()

    def create_model(self):
        self.model = Sequential()
        if self.image_deformation:
            self.model.add(Reshape(self.reshape, input_shape=self
               .shape))
        self.model.add(Dense(self.hidden_neurons, activation='
           sigmoid', input_shape=self.reshape))
        self.model.add(Dense(self.num_classes, activation='
           softmax'))

        self.model.compile(loss='mean_squared_error',
            optimizer=SGD(lr=self.learning_rate),
            metrics=['accuracy'])

    def train(self):
        (x_train, y_train), (x_test, y_test) = cifar10.load_data
           ()
```

```
y_train = to_categorical(y_train, self.num_classes)
y_test = to_categorical(y_test, self.num_classes)

if not self.image_deformation:
    x_train = x_train.reshape(x_train.shape[0], 3072)
    x_test = x_test.reshape(x_test.shape[0], 3072)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

if self.image_deformation:
    datagen = ImageDataGenerator(
        width_shift_range=0.1,
        height_shift_range=0.1,
    )
    datagen.fit(x_train)

    history = self.model.fit_generator(datagen.flow(
       x_train, y_train,
            batch_size=self.batch_size),
        steps_per_epoch=x_train.shape[0] // self.
           batch_size,
        verbose=1,
        epochs=self.n_iterations,
        validation_data=(x_test, y_test))
else:
    history = self.model.fit(x_train, y_train,
        batch_size=self.batch_size,
        epochs=self.n_iterations,
        verbose=1,
        validation_data=(x_test, y_test))
```

```
        result = [(history.history.get('acc')[i], history.history
            .get('loss')[i],
                      history.history.get('val_acc')[i], history.
                         history.get('val_loss')[i])
                          for i in range(len(history.history.get('
                             acc')))]
        result_string = 'position,train_accuracy,train_loss,
            validation_accuracy,validation_loss'
        for position in range(len(result)):
            result_string += '\n{:d},{:1.5f},{:1.5f},{:1.5f
                },{:1.5f}'.format(
                 position, result[position][0], result[position
                    ][1], result[position][2], result[position
                    ][3])

        return result_string
```

A.2. CNN

La arquitectura CNN, por otra parte, no ha necesitado aplanar la entrada en caso de deformación pues reciben una entrada bi o tridimensional directamente. Sin embargo, ha tenido que realizar el proceso contrario, es decir, que si no se realizan deformaciones se convierta –en MNIST– de vector unidimensional a matriz de tres dimensiones –ancho, alto y profundidad, que es 1 al ser solo blanco y negro–.

A.2.1. MNIST

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras import backend as K
from keras.losses import categorical_crossentropy
```

```
from keras.optimizers import rmsprop
from keras.utils import to_categorical
from keras.datasets import mnist


class Cnn:
    num_classes = 10
    batch_size = 128
    # Image dimension
    img_rows, img_cols = 28, 28

    def __init__(self, n, hidden_layer, learning_rate,
       dropout_probability):
        self.n_iterations = n
        self.hidden_neurons = hidden_layer
        self.learning_rate = learning_rate
        self.dropout_probability = dropout_probability
        self.create_model()

    def create_model(self):
        if K.image_data_format() == 'channels_first':
            input_shape = (1, self.img_rows, self.img_cols)
        else:
            input_shape = (self.img_rows, self.img_cols, 1)

        self.model = Sequential()
        self.model.add(Conv2D(32, kernel_size=(3, 3),
                              activation='relu',
                              input_shape=input_shape))
        self.model.add(Conv2D(32, (3, 3), padding='same',
           activation='relu'))
        self.model.add(MaxPooling2D(pool_size=(2, 2)))
        self.model.add(Dropout(self.dropout_probability))
```

```
        self.model.add(Conv2D(64, (3, 3), padding='same',
            activation='relu'))
        self.model.add(Conv2D(64, (3, 3), activation='relu'))
        self.model.add(MaxPooling2D(pool_size=(2, 2)))
        self.model.add(Dropout(self.dropout_probability))

        self.model.add(Flatten())
        self.model.add(Dense(self.hidden_neurons, activation='
            relu'))
        self.model.add(Dropout(self.dropout_probability))
        self.model.add(Dense(self.hidden_neurons, activation='
            relu'))
        self.model.add(Dropout(self.dropout_probability))
        self.model.add(Dense(self.num_classes, activation='
            softmax'))

        self.model.compile(loss=categorical_crossentropy,
                           optimizer=rmsprop(lr=self.
                               learning_rate, decay=1e-6),
                           metrics=['accuracy'])

    def train(self):
        (x_train, y_train), (x_test, y_test) = mnist.load_data()

        if K.image_data_format() == 'channels_first':
            x_train = x_train.reshape(x_train.shape[0], 1, self.
                img_rows, self.img_cols)
            x_test = x_test.reshape(x_test.shape[0], 1, self.
                img_rows, self.img_cols)
        else:
            x_train = x_train.reshape(x_train.shape[0], self.
                img_rows, self.img_cols, 1)
```

```
        x_test = x_test.reshape(x_test.shape[0], self.
            img_rows, self.img_cols, 1)

    x_train = x_train.astype('float32')
    x_test = x_test.astype('float32')
    x_train /= 255
    x_test /= 255

    # convert class vectors to binary class matrices
    y_train = to_categorical(y_train, self.num_classes)
    y_test = to_categorical(y_test, self.num_classes)

    datagen = ImageDataGenerator(
        width_shift_range=0.1,
        height_shift_range=0.1,
    )
    datagen.fit(x_train)

    history = self.model.fit_generator(datagen.flow(x_train,
        y_train,
             batch_size=self.batch_size),
        steps_per_epoch=x_train.shape[0] // self.batch_size,
        verbose=1,
        epochs=self.n_iterations,
        validation_data=(x_test, y_test))
    result = [(history.history.get('acc')[i], history.history
        .get('loss')[i],
                history.history.get('val_acc')[i], history.
                    history.get('val_loss')[i])
                 for i in range(len(history.history.get('
                     acc')))]
    result_string = 'position,train_accuracy,train_loss,
        validation_accuracy,validation_loss'
```

```
        for position in range(len(result)):
            result_string += '\n{:d},{:1.5f},{:1.5f},{:1.5f
               },{:1.5f}'.format(
                position, result[position][0], result[position
                   ][1], result[position][2], result[position
                   ][3])

        return result_string
```

A.2.2. CIFAR

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.losses import categorical_crossentropy
from keras.utils import to_categorical
from keras.optimizers import rmsprop
from keras.datasets import cifar10


class Cnn:
    num_classes = 10
    batch_size = 128
    # Image dimension
    img_rows, img_cols = 28, 28
    shape = (32, 32, 3)

    def __init__(self, n, hidden_layer, learning_rate,
       dropout_probability):
        self.n_iterations = n
        self.hidden_neurons = hidden_layer
        self.learning_rate = learning_rate
        self.dropout_probability = dropout_probability
```

```
        self.create_model()

    def create_model(self):
        self.model = Sequential()
        self.model.add(Conv2D(32, (3, 3), padding='same',
            input_shape=self.shape,
            activation='relu'))
        self.model.add(Conv2D(32, (3, 3), activation='relu'))
        self.model.add(MaxPooling2D(pool_size=(2, 2)))
        self.model.add(Dropout(self.dropout_probability))

        self.model.add(Conv2D(64, (3, 3), padding='same',
           activation='relu'))
        self.model.add(Conv2D(64, (3, 3), activation='relu'))
        self.model.add(MaxPooling2D(pool_size=(2, 2)))
        self.model.add(Dropout(self.dropout_probability))

        self.model.add(Flatten())
        self.model.add(Dense(self.hidden_neurons, activation='
           relu'))
        self.model.add(Dropout(self.dropout_probability))
        self.model.add(Dense(self.hidden_neurons, activation='
           relu'))
        self.model.add(Dropout(self.dropout_probability))
        self.model.add(Dense(self.num_classes, activation='
           softmax'))

        # Let's train the model using RMSprop
        self.model.compile(loss=categorical_crossentropy,
            optimizer=rmsprop(lr=self.learning_rate, decay=1e-6),
            metrics=['accuracy'])

    def train(self):
```

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data
    ()

# convert class vectors to binary class matrices
y_train = to_categorical(y_train, self.num_classes)
y_test = to_categorical(y_test, self.num_classes)

x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

datagen = ImageDataGenerator(
    width_shift_range=0.1,
    height_shift_range=0.1,
)
datagen.fit(x_train)

history = self.model.fit_generator(datagen.flow(x_train,
    y_train,
         batch_size=self.batch_size),
    steps_per_epoch=x_train.shape[0] // self.batch_size,
    verbose=1,
    epochs=self.n_iterations,
    validation_data=(x_test, y_test))
result = [(history.history.get('acc')[i], history.history
    .get('loss')[i],
             history.history.get('val_acc')[i], history.
                history.get('val_loss')[i])
                 for i in range(len(history.history.get('
                    acc')))]
result_string = 'position,train_accuracy,train_loss,
    validation_accuracy,validation_loss'
```

```
    for position in range(len(result)):
        result_string += '\n{:d},{:1.5f},{:1.5f},{:1.5f
           },{:1.5f}'.format(
            position, result[position][0], result[position
               ][1], result[position][2], result[position
               ][3])

    return result_string
```